Czech Technical University in Prague
Faculty of Electrical Engineering

Symbolic Machine Learning Exam-Oriented Skripta

Thirty-fourth draft in English

Prepared from the uploaded presentations, tutorial sheets, model exam, and course pages.
This is an original study text, not an official CTU textbook.

May 25, 2026

Preface

This thirty-fourth draft is reorganized according to the presentation flow: first the non-NLP material, then the NLP block. It also includes the first lecture on Markov processes, Markov reward processes, and Markov decision processes. The text is intentionally not a rigorous mathematical monograph. The goal is to explain the ideas well enough for exam preparation and for practical understanding of the algorithms.

The structure is still inspired by CTU-style skripta: chapters, definitions, examples, exercises, and a table of contents. Compared with the previous drafts, this version adds more solved examples, more diagrams, more exam tips, and more explicit links between similar ideas. It also keeps an "ELI5" paragraph for important concepts.

> **Takeaway.** Read the formulas as compact descriptions of algorithms, not as proof obligations. For this course, it is usually more important to know what the object means, how it is used, and what can go wrong than to reproduce a long derivation.

How this draft is ordered

1. Lectures 1–6: reinforcement learning, function approximation, and bandits.
2. Computational learning theory.
3. NLP lectures: probabilistic, vector, matrix, and neural models.
4. Exam checklists and self-review.

How to use it

For each concept, focus on three pieces of information: the problem it solves, the central formula or algorithm, and the typical exam trap. If these three pieces are clear, the concept is usually understood at the level expected by a written exam.

In this version, transitions between sections are made more explicit. Many sections now end by saying what problem remains and why the next section exists. This is intentional: the course topics are not isolated islands. For example, function approximation leads naturally to the question of training targets, and Monte Carlo and TD methods are two different answers to that question.

Examples are now consistently typeset as numbered *Example* blocks. Clarifications that came from common points of confusion are written as normal explanatory text instead of as unmotivated question–answer fragments.

Cross-reference map

This draft now contains many clickable internal references. To avoid making formulas unreadable, single-letter symbols such as P, R, V, and Q are not linked after every occurrence. Instead, important sections include a short "Referenced concepts" note, and the table below gives the canonical place where each term is explained.

Term or symbol	Main place to read about it
Markov property	Section 1.1
transition model P	Section 1.2; for MDPs also Section 1.7; empirical estimates in Section 1.8
reward R, observed reward r_t	Section 1.3; for MDP rewards Section 1.7; empirical estimates in Section 1.8
counting notation $N(s)$, $N(s,a)$	Section 1.8
return G_t or g_t	Section 1.4
discount factor γ	Section 1.5.1
state-value function V or V^π	Sections 1.5 and 1.11
Bellman equation	Sections 1.6 and 2.1
MDP	Section 1.7
policy π	Section 1.9
MDP + fixed policy = MRP	Section 1.10
action-value function Q^π and learned estimate $Q(s,a)$	Section 1.12; bridge to control in Section 3.6
policy evaluation	Section 2.1
policy improvement	Section 2.2
Monte Carlo evaluation	Section 3.2
TD learning	Section 3.3
learning-rate conditions	Section 3.5
SARSA	Section 4.5
Q-learning	Section 4.6
on-policy/off-policy	Section 4.3
GLIE	Section 4.8
function approximation	Section 5.1
deep Q-learning / DQN	Section 5.8
features	Section 5.2
bandit action value and regret	Section 6.2
concept learning	Section 7.2
learning models	Section 7.3
mistake-bound model	Section 7.4
conjunction learner, halving, Winnow	Section 7.5
reductions and improper learning	Sections 7.6 and 7.6.3
PAC learning and assumptions	Sections 7.7 and 7.7.2
finite hypothesis class PAC bound	Section 7.7.5.1
VC dimension	Section 7.8
NLP chapter map	Section 8.1
language model	Section 8.3
N-gram model	Section 8.5
perplexity	Section 8.6
Naive Bayes	Section 8.7

TF-IDF	Section 9.3
embeddings	Section 9.6
skip-gram	Section 9.5
topic models	Section 10.2
LSA/SVD	Section 10.3
NMF	Section 10.4
LDA	Section 10.5
topic models versus LLMs	Section 10.6
RNN	Section 11.5
Transformer	Section 11.7
query/key/value attention	Section 11.7.4

Contents

Part I

Reinforcement Learning and Bandits

Chapter 1

Lecture 1: Markov Processes, Rewards, and Decisions

1.1 Random processes and the Markov property

A random process is a sequence of random variables

$$X_1, X_2, X_3, \ldots$$

where X_t is the state of the process at time t. The capital letter X_t denotes the random variable; a lower-case symbol such as s denotes one concrete state value that X_t may take.

Symbol guide.

- S: state space, the set of all possible states.
- X_t: random variable representing the state at time t.
- s, s': concrete states, usually elements of S.
- $P(s' \mid s)$: probability that the next state is s' if the current state is s.

Definition 1.1 (Markov property)**.**
A process has the Markov property if the distribution of the next state depends only on the current state and not on the full history:

$$\mathbb{P}(X_{t+1} = s' \mid X_t = s, X_{t-1}, \ldots, X_1) = \mathbb{P}(X_{t+1} = s' \mid X_t = s).$$

In words: the present state contains all information needed for predicting the next-state distribution.

ELI5. Imagine a board game. A state can be a square on the board. If knowing the current state – the square you are standing on – is enough to know the possible next states, then you do not need the whole story of how you got there. That is the Markov property.

If a process is not Markov, one common repair is to enlarge the state. For example, if the next behavior of a robot depends on both its position and whether it carries a package, then the state should contain both the position and the package status.

Exam use. When checking the Markov property, test whether the current state contains all information needed for predicting the next transition. If not, state what information must be added to the state.

1.2 Transition probabilities and transition matrices

For a finite state space, transition probabilities can be stored in a transition matrix. If $S = \{s_1, \ldots, s_n\}$, then a matrix entry stores a number such as

$$P(s_i \mid s_j) = \mathbb{P}(X_{t+1} = s_i \mid X_t = s_j).$$

For each current state, the probabilities of all possible next states must sum to one.

Example 1.1 (Three-state weather process)**.**
Let $S = \{\text{sun}, \text{cloud}, \text{rain}\}$. The number

$$P(\text{rain} \mid \text{cloud}) = 0.4$$

means: if today is cloudy, tomorrow is rainy with probability 0.4. Once today's weather is known, the Markov model ignores whether yesterday was sunny or rainy.

1.3 Markov reward process: reward convention

Referenced concepts. Used here: transition model P (Section 1.2); discount factor γ (Section 1.5.1).

A Markov reward process, abbreviated MRP, adds rewards to a Markov process. In the lecture convention, the reward depends on the state. There are no actions yet.

Definition 1.2 (Markov reward process)**.**
An MRP consists of:

- a state space S;
- transition probabilities $P(s' \mid s)$;
- a reward function $R(s) = \mathbb{E}[R_t \mid X_t = s]$;
- a discount factor $\gamma \in [0, 1]$.

Symbol guide.

- R_t: random reward received at time t.
- r_t: one observed numeric reward in one sampled episode.
- $R(s)$: expected immediate reward when the process is in state s.
- γ: discount factor; controls how much future rewards count.

In an MRP, rewards are assigned according to the state you are in, using the reward convention of the lecture. If $R(5) = 10$, then being in state 5 gives expected reward 10. There is no action choice in an MRP, so rewards cannot depend on the action.

Some books put rewards on transitions, for example "reward for entering state s'". That is just a different convention. In this course's MRP notation, the main object is $R(s)$, an expected reward attached to the current state.

ELI5. An MRP is like a board game where each state is a square on the board. Some squares give points when you stand on them. You do not choose moves; the rules randomly move you. Your score comes from the states, i.e. board squares, that you visit.

1.4 Return: the total future reward

Referenced concepts. Used here: reward R / r_t (Section 1.3); discount factor γ (Section 1.5.1).

The return from time t is the discounted sum of rewards from that time onward:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots .$$

G_t is a random variable, because future states and rewards are random.

For one sampled episode, we often write the realized return as a lower-case number g_t:

$$g_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots .$$

Symbol guide.

- G_t: random return from time t.
- g_t: realized numeric return computed from one episode.
- Units: same kind of units as reward, e.g. points, money, clicks, cost.

1.5 State-value function in an MRP

Referenced concepts. Used here: reward R / r_t (Section 1.3); return G_t (Section 1.4); discount factor γ (Section 1.5.1); Bellman equation (Sections 1.6, 2.1).

The state-value function is

$$V(s) = \mathbb{E}[G_t \mid X_t = s].$$

It maps each state to one real number. That number is not an immediate reward. It is the expected discounted future reward when starting in that state.

Symbol guide.

- $V : S \to \mathbb{R}$: function from states to real-valued expected returns.
- $V(s)$: one scalar number, the value of state s.
- Output type of a Bellman equation: one value per state; for finite S, a vector in $\mathbb{R}^{|S|}$.

ELI5. The reward of a state is what you get immediately in that state. If states are imagined as squares on a board, the reward is the points printed on the square right now. The value of a state is bigger-picture: how good it is to be in that state when we also count the future points we will probably collect after moving from it.

1.5.1 Discount factor

The discount factor has two roles:

- practically, it says that earlier rewards matter more;
- technically, it can keep infinite sums finite.

Special cases:

$$\gamma = 0 \quad \text{only immediate reward matters,}$$

$$\gamma \approx 1 \quad \text{future rewards matter a lot.}$$

1.6 Bellman equation for an MRP

Referenced concepts. Used here: transition model P (Section 1.2); reward R / r_t (Section 1.3); return G_t (Section 1.4); discount factor γ (Section 1.5.1).

For an MRP, we know the environment model – P (transition model), R (reward function), and γ (discount factor) – and want to compute the value of every state. The MRP Bellman equation is

$$V(s) = R(s) + \gamma \sum_{s' \in S} P(s' \mid s) V(s').$$

Read the formula from left to right:

- $V(s)$ is the unknown value of the current state s (one expected long-term score).
- $R(s)$ is the expected immediate reward in state s (what you get now).
- $P(s' \mid s)$ is the probability that the next state will be s' if the current state is s.
- $V(s')$ is the value of that possible next state.
- $\sum_{s' \in S} P(s' \mid s) V(s')$ is an expectation: average future value over all possible next states.
- γ discounts that future value (how much future rewards count).

In words:

$$\text{value now} = \text{reward now} + \text{discounted average value of where we go next.}$$

Symbol guide.

- Inputs to the equation: P (transition model), R (reward function), γ (discount factor).
- Unknown: $V(s)$ for every state $s \in S$.
- Output after solving: a value table/vector, one real number for each state.
- Type of output: for finite S, $V \in \mathbb{R}^{|S|}$; for example, if there are three states, the output is three numbers.

The important point is that this is not one isolated equation. It is one equation for each state, and the equations usually depend on each other. A state's value may depend on another state's value, and sometimes even on its own value, because the process can return to the same state later.

Example 1.2 (Library and cafe MRP)**.**
This example is intentionally small, because the point is to see what the Bellman equation is doing.

Story. A student's day is described only by where the student is during a time step. One time step can be imagined as one hour. There are two states:

$$S = \{L, C\},$$

where L means "library" and C means "cafe".

Immediate rewards. The reward is received for being in the current state during the current time step:

$$R(L) = 1, \qquad R(C) = 4.$$

So the library gives 1 point now, while the cafe gives 4 points now. This does not yet say anything about the future.

Transitions. After the current time step, the student may stay or move:

$$P(L \mid L) = 0.7, \quad P(C \mid L) = 0.3,$$

$$P(L \mid C) = 0.5, \quad P(C \mid C) = 0.5.$$

For example, $P(C \mid L) = 0.3$ means: if the current state is library, the next state is cafe with probability 0.3. Let $\gamma = 0.5$.

Computed quantities. We want two unknown numbers:

$$V(L) \quad \text{and} \quad V(C).$$

They are not immediate rewards. They are long-term expected discounted returns from the two starting states.

Bellman equation for the library. Start in L:

$$V(L) = R(L) + \gamma\big(P(L \mid L)V(L) + P(C \mid L)V(C)\big).$$

Now substitute the actual numbers:

$$V(L) = 1 + 0.5\big(0.7V(L) + 0.3V(C)\big).$$

Read this as: get 1 point now; then the future is a weighted average of the value of the possible next states.

Bellman equation for the cafe. Start in C:

$$V(C) = R(C) + \gamma\big(P(L \mid C)V(L) + P(C \mid C)V(C)\big),$$

therefore

$$V(C) = 4 + 0.5\big(0.5V(L) + 0.5V(C)\big).$$

Solving. These two equations must be true at the same time. Solving them gives

$$V(L) = 3, \qquad V(C) = \frac{19}{3} \approx 6.33.$$

The cafe value is larger than the immediate reward 4 because it includes future rewards. The library value is larger than the immediate reward 1 because from the library the student may later reach the cafe.

Example 1.3 (One-state sanity check)**.**
If there is only one state s, reward $R(s) = 2$, and the process always returns to s, then

$$V(s) = 2 + \gamma V(s).$$

For $\gamma = 0.5$, $V(s) = 2 + 0.5V(s)$, so $V(s) = 4$. This means: immediate reward is 2, but the expected discounted sum of repeated future rewards is 4.

> **Common trap.** Do not confuse reward (see Section 1.3), return (see Section 1.4), and value (see Sections 1.5 and 1.11). Reward is one-step. Return is a whole future discounted sum. Value is the expected return from a state.

1.7 Markov decision process: adding actions

Referenced concepts. Used here: transition model P (Section 1.2); reward R / r_t (Section 1.3); discount factor γ (Section 1.5.1).

A Markov decision process, abbreviated MDP, adds actions. The agent can influence transitions and rewards by choosing an action.

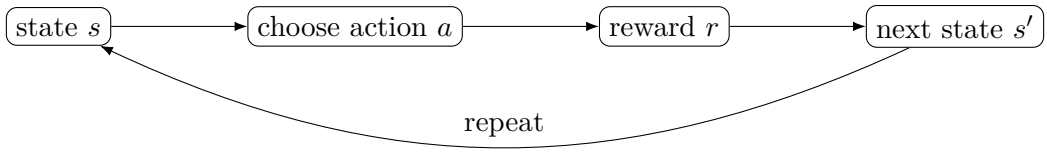
Definition 1.3 (Markov decision process)**.**
An MDP consists of:

- states S;
- actions A;
- transition probabilities $P(s' \mid s, a)$;
- expected rewards $R(s, a)$;
- discount factor γ.

Symbol guide.

- A: set of possible actions.
- A_t: random action at time t.
- a: one concrete action from A.
- $P(s' \mid s, a) = \mathbb{P}(X_{t+1} = s' \mid X_t = s, A_t = a)$: transition model.
- $R(s, a) = \mathbb{E}[R_t \mid X_t = s, A_t = a]$: expected immediate reward after choosing action a in state s.

So in an MDP, the reward usually depends on what state you are in and what action you take. In some diagrams, a reward is drawn on entering a terminal state or on a transition. That can be converted into this notation by defining $R(s, a)$ as the expected reward caused by taking action a in state s.



ELI5. An MDP is like a game. You stand in a position, choose a move, get points, and land in a new position. Different moves can give different points and lead to different next positions.

1.8 Empirical model estimates from samples

Referenced concepts. Used here: transition model P (Sections 1.2, 1.7); reward function R (Sections 1.3, 1.7); observed reward r_t (Section 1.3).

The previous sections defined the *true* transition and reward models. For example, $P(s' \mid s)$ is the real probability of moving from s to s', and $R(s)$ is the true expected immediate reward in state s. In exam problems or tutorials, however, we may only see sampled experience. Then we can estimate these objects by counting transitions and averaging rewards.

Symbol guide.

- $N(s)$: number of observed time steps where the current state was s.
- $N(s, s')$: number of observed transitions from s to s'.
- $N(s, a)$: number of observed time steps where the agent was in s and took action a.
- $N(s, a, s')$: number of observed transitions where the agent was in s, took action a, and landed in s'.
- A hat, as in $\widehat{P}$ or $\widehat{R}$, means an empirical estimate from data, not the exact model.

For a Markov process or MRP without actions, the empirical transition estimate is

$$\widehat{P}(s' \mid s) = \frac{N(s, s')}{N(s)}.$$

This means: out of all observed visits to state s, count the fraction where the next state was s'.

For an MDP with actions, we also condition on the action:

$$\widehat{P}(s' \mid s, a) = \frac{N(s, a, s')}{N(s, a)}.$$

This means: out of all observed times where we took action a in state s, count the fraction where the next state was s'.

Rewards are estimated by averages. In an MRP,

$$\widehat{R}(s) = \frac{1}{N(s)} \sum_{t:s_t=s} r_t,$$

and in an MDP,

$$\widehat{R}(s, a) = \frac{1}{N(s, a)} \sum_{t:s_t=s,\ a_t=a} r_t.$$

So N is not part of the definition of an MDP or MRP. It appears only when we estimate an unknown model from observed samples.

Example 1.4 (Estimating a transition probability)**.**
Suppose that from state L we observed 10 transitions in total. In 7 of them the next state was again L, and in 3 of them the next state was C. Then

$$N(L) = 10, \qquad N(L, L) = 7, \qquad N(L, C) = 3,$$

so

$$\widehat{P}(L \mid L) = \frac{7}{10}, \qquad \widehat{P}(C \mid L) = \frac{3}{10}.$$

Common trap. Do not confuse exact model notation and empirical estimate notation. $P(s' \mid s, a)$ is the true transition model. $\widehat{P}(s' \mid s, a)$ is our estimate computed from counts such as $N(s, a, s')/N(s, a)$.

1.9 Policy

A policy tells the agent how to act.

Definition 1.4 (Policy)**.**
A stochastic policy is written

$$\pi(a \mid s) = \mathbb{P}(A_t = a \mid X_t = s).$$

For each state s, $\pi(\cdot \mid s)$ is a probability distribution over actions:

$$\sum_{a \in A} \pi(a \mid s) = 1.$$

A deterministic policy chooses one action in each state.

Symbol guide.

- π: policy.
- $\pi(a \mid s)$: probability that the policy chooses action a in state s.
- Deterministic shorthand: $\pi(s) = a$ means action a is always chosen in state s.

ELI5. A policy is the player's strategy. It says what the player does in every situation. It can be strict, like "always go right", or random, like "go right with probability 0.7 and left with probability 0.3".

1.10 MDP plus policy equals MRP

Referenced concepts. Used here: reward R / r_t (Section 1.3); discount factor γ (Section 1.5.1); MDP (Section 1.7); policy π (Section 1.9).

Once a policy is fixed, the agent's action choice is no longer an open decision problem. At each state, the policy randomly picks actions according to fixed probabilities. Therefore an MDP under a fixed policy becomes an MRP:

$$P^\pi(s' \mid s) = \sum_{a \in A} \pi(a \mid s) P(s' \mid s, a),$$

$$R^\pi(s) = \sum_{a \in A} \pi(a \mid s) R(s, a).$$

Symbol guide.

- $P^\pi(s' \mid s)$: transition probability after averaging over actions selected by policy π.
- $R^\pi(s)$: expected immediate reward in state s after averaging over actions selected by π.
- Resulting MRP: $(S, P^\pi, R^\pi, \gamma)$.

ELI5. Before choosing a strategy, the MDP still contains open action choices. After choosing a fixed strategy, the policy automatically selects actions according to its probabilities. The remaining system is again only a random process with rewards: an MRP.

Example 1.5 (Averaging rewards under a policy)**.**
Suppose in state s there are two actions. The policy chooses a_1 with probability 0.7 and a_2 with probability 0.3. If $R(s, a_1) = 10$ and $R(s, a_2) = 0$, then

$$R^\pi(s) = 0.7 \cdot 10 + 0.3 \cdot 0 = 7.$$

The policy-induced state reward is 7, even though neither action has reward exactly 7. It is an expectation.

1.11 State-value function in an MDP

Referenced concepts. Used here: reward R / r_t (Section 1.3); return G_t (Section 1.4); discount factor γ (Section 1.5.1); Bellman equation (Sections 1.6, 2.1); policy π (Section 1.9).

For a fixed policy π, the return is

$$G_t^\pi = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots$$

where future actions are chosen according to π. The state-value function is

$$V^\pi(s) = \mathbb{E}[G_t^\pi \mid X_t = s].$$

Symbol guide.

- $V^\pi : S \to \mathbb{R}$.
- $V^\pi(s)$: expected discounted future reward from state s if we follow policy π.
- It is not an action, not a probability, and not only the immediate reward.

The Bellman equation for evaluating a policy is

$$V^\pi(s) = \sum_{a \in A} \pi(a \mid s) \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^\pi(s') \right].$$

Solving this equation for all states outputs the value function V^π, i.e. a real number for every state.

ELI5. $V^\pi(s)$ is the expected long-term score of state s when you promise to follow policy π. It is not the reward received immediately in s. It is an average over possible futures: start in s, let the policy choose actions, let the environment randomly move, add all future rewards with discounting.

1.12 Action-value function: definition and role of Q

Referenced concepts. Used here: reward R / r_t (Section 1.3); return G_t (Section 1.4); discount factor γ (Section 1.5.1); state-value function V, V^π (Sections 1.5, 1.11); policy π (Section 1.9); evaluation vs control (Section 4.1).

The state-value function $V^\pi(s)$ tells us how good a state is under a fixed policy. But if we want to improve a policy, we must compare actions. That is why we define the action-value function Q^π.

Definition 1.5 (Action-value function)**.**

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^\pi(s').$$

Symbol guide.

- $Q^\pi : S \times A \to \mathbb{R}$.

- $Q^\pi(s, a)$: expected discounted return if we first force action a in state s, and after that follow policy π.
- Same units as reward and value: expected discounted reward.

ELI5. $V^\pi(s)$ represents how good state s is if the agent follows policy π. If states are board squares, it is the score of standing on that square under the strategy. $Q^\pi(s, a)$ represents how good it is to first take action a in state s and only then continue with policy π.

This is also why Q is useful for control: in one fixed state s, compare all numbers $Q^\pi(s, a)$ and choose the action with the largest value. In other words, V^π grades states, while Q^π grades first actions available in a state.

Exam use. If you see $Q^\pi(s, a)$, read it as "first action value". It is not a new environment parameter. It is computed from R, P, γ, and V^π. Later, in SARSA and Q-learning, the superscript is often omitted and $Q(s, a)$ means the learner's current estimate stored in a table.

Chapter 2

Lecture 1 Continued: Dynamic Programming Control

2.1 Policy evaluation

> **Referenced concepts.** Used here: transition model P (Section 1.2); reward R / r_t (Section 1.3); return G_t (Section 1.4); discount factor γ (Section 1.5.1); state-value function V, V^π (Sections 1.5, 1.11); policy π (Section 1.9).

Policy evaluation means computing V^π for a fixed policy π. In this step, the strategy is already chosen. We are not yet asking for a better strategy; we are only asking how good the current one is.

If the model is known, the inputs are:

- $P(s' \mid s, a)$: transition model; probability of next state s' after action a in state s;
- $R(s, a)$: reward function; expected immediate reward after action a in state s;
- γ: discount factor; how strongly future rewards count;
- $\pi(a \mid s)$: fixed policy; probability that the agent chooses action a in state s.

Then the Bellman equation can be solved or iterated:

$$V^\pi(s) = \sum_a \pi(a \mid s) \left[R(s, a) + \gamma \sum_{s'} P(s' \mid s, a) V^\pi(s') \right].$$

Read the formula as follows: in state s, the policy may choose different actions a. For each possible action, we take immediate reward $R(s, a)$ plus discounted expected future value. Then we average this over actions according to $\pi(a \mid s)$.

> **Symbol guide.**
>
> - Input: P (transition model), R (reward function), γ (discount factor), and π (fixed policy).
> - Unknown/output: V^π (value function of the fixed policy).
> - Type of output: for finite S, a vector in $\mathbb{R}^{|S|}$, one scalar expected return for each state.
> - Meaning of $V^\pi(s)$: expected discounted return from state s when future actions are chosen by π.

ELI5. A strategy has already been fixed. Policy evaluation does not improve it yet; it only estimates how good each state is when that fixed strategy is kept.

Common trap. The Bellman equation does not output a reward function. It outputs a value function. Rewards are inputs; values are long-term expected consequences of those rewards.

2.2 Policy improvement

Referenced concepts. Used here: reward R / r_t (Section 1.3); return G_t (Section 1.4); discount factor γ (Section 1.5.1); state-value function V, V^π (Sections 1.5, 1.11); policy π (Section 1.9); action-value function Q^π (Section 1.12).

After evaluating policy π, we can improve it by comparing first actions using Q^π:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s').$$

Then choose a greedy action:

$$\pi'(s) \in \operatorname{argmax}_{a \in A} Q^\pi(s,a).$$

The notation $\operatorname{argmax}_a Q^\pi(s,a)$ means: among all actions a, return the action whose Q value is largest. It returns an action, not the value itself. The largest value would be written $\max_a Q^\pi(s,a)$.

ELI5. First you grade the old strategy. Then, for each state, compare each possible first move while assuming that after this first move the old strategy continues. The best first move becomes the improved policy for that state.

Example 2.1 (Computing Q-values from V)**.**
Suppose we are in state s and compare two first actions. Let $\gamma = 0.5$.

Action `left` gives immediate reward $R(s, \texttt{left}) = 0$ and deterministically moves to a state s_L whose old-policy value is $V^\pi(s_L) = 6$:

$$Q^\pi(s, \texttt{left}) = 0 + 0.5 \cdot 6 = 3.$$

Action `right` gives immediate reward $R(s, \texttt{right}) = 2$ and deterministically moves to a state s_R whose old-policy value is $V^\pi(s_R) = 3$:

$$Q^\pi(s, \texttt{right}) = 2 + 0.5 \cdot 3 = 3.5.$$

Then the improved greedy policy chooses

$$\pi'(s) = \texttt{right}.$$

The value 3.5 is the largest score; the action `right` is the argmax.

2.3 Policy iteration

Referenced concepts. Used here: policy π (Section 1.9); policy evaluation (Section 2.1); policy improvement (Section 2.2).

Policy iteration alternates two steps:

1. policy evaluation: compute V^{π_i} for the current policy;

2. policy improvement: compute greedy actions using Q^{π_i}.

When the policy no longer changes, it is optimal for finite discounted MDPs.

```
initialize a policy pi randomly
repeat
    evaluate V^pi
    for each state s:
        compute Q^pi(s,a) for all actions a
        pi_new(s) = argmax_a Q^pi(s,a)
    pi = pi_new
until policy does not change
```

ELI5. Policy iteration is like improving a route plan. First calculate how good the current route plan is. Then change each decision to the best local choice according to that calculation. Repeat until nothing changes.

2.4 Value iteration

Referenced concepts. Used here: discount factor γ (Section 1.5.1); state-value function V, V^π (Sections 1.5, 1.11); Bellman equation (Sections 1.6, 2.1); policy π (Section 1.9); policy iteration (Section 2.3); function approximation (Section 5.1).

Value iteration does not fully evaluate each intermediate policy. It repeatedly applies the optimal Bellman backup:

$$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_k(s') \right].$$

After convergence, extract a greedy policy:

$$\pi(s) \in \operatorname{argmax}_a \left[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \right].$$

Symbol guide.

- V_k: current approximation of the optimal value function at iteration k.
- V_{k+1}: improved approximation after one Bellman backup.
- max: returns the best value.
- argmax: returns an action achieving the best value.

ELI5. Policy iteration says: fully grade one strategy, then improve the strategy. Value iteration says: keep directly updating the score of each state as if you will choose the best next action.

Takeaway. Policy iteration improves policies explicitly. Value iteration improves value estimates directly. Both need the model $P(s' \mid s,a)$ and $R(s,a)$.

2.5 What to know for the exam

Referenced concepts. Used here: transition model P (Section 1.2); reward R / r_t (Section 1.3); return G_t (Section 1.4); state-value function V, V^π (Sections 1.5, 1.11); Bellman equation (Sections 1.6, 2.1); MDP (Section 1.7); policy π (Section 1.9).

You should be able to:

- explain the difference between reward (Section 1.3), return (Section 1.4), and value (Sections 1.5, 1.11);
- explain what P (transition model; Sections 1.6, 1.7), R (reward function; Sections 1.3, 1.7), π (policy; Section 1.9), V^π (state-value function; Section 1.11), and Q^π (action-value function; Section 1.12) mean;
- write the Bellman equation for V^π and know what its inputs and output are (Section 2.1);
- explain why MDP + fixed policy becomes MRP (Section 1.10);
- explain policy improvement and the meaning of argmax (Section 2.2);
- distinguish policy iteration (Section 2.3) and value iteration (Section 2.4);
- compute one simple Bellman update (Section 1.6) or TD update (Section 3.3) by hand.

Chapter 3

Lecture 2: Model-Free Policy Evaluation

3.1 Motivation for model-free methods

Referenced concepts. Used here: transition model P (Section 1.2); reward R / r_t (Section 1.3); return G_t (Section 1.4); discount factor γ (Section 1.5.1); state-value function V, V^π (Sections 1.5, 1.11); policy π (Section 1.9); policy evaluation (Section 2.1).

Dynamic programming methods assume that we know transition probabilities and rewards. In many real situations we do not know them, but we can interact with the environment and observe episodes.

Definition 3.1 (Model-free policy evaluation)**.**
Given a fixed policy π, estimate V^π from sampled interaction data without knowing the full model $P(s' \mid s, a)$ or $R(s, a)$.

Symbol guide.

- π: the fixed policy being evaluated. It tells the agent which actions to take. In this chapter, π is *not* being learned yet.
- $P(s' \mid s, a)$: transition model, the probability that action a in state s leads to next state s'.
- $R(s, a)$: reward function, the expected immediate reward after doing action a in state s.
- γ: discount factor, how much future rewards count.
- $V^\pi(s)$: value of state s under policy π, i.e. expected discounted future return when starting in s and then following π.
- Goal/output: estimates $\widehat{V}^\pi(s)$, one real number per state.

Common trap. Sections 3.2 and 3.3 are evaluation methods, not control methods. They do not create a new policy by themselves. They estimate how good each state is under the fixed policy π Policy creation starts in the next chapter, where we learn or improve $Q(s, a)$, the current estimate of the action-value function (Section 1.12; also see the bridge in Section 3.6).

ELI5. Imagine you already chose a strategy for a game, for example "always go right unless blocked". Policy evaluation does not invent a better strategy. It only watches many games played with that strategy and estimates how good each position is.

Takeaway. Roadmap of this chapter. We want to estimate values for a fixed policy from experience. Monte Carlo evaluation answers this by waiting until an episode finishes and then using the observed return as a sample target. TD learning answers the same problem by updating immediately after one transition, using the current estimate of the next state. Both are still evaluation methods. Control, where the policy is improved, starts in the next chapter.

3.2 Monte Carlo policy evaluation

Referenced concepts. Used here: return G_t (Section 1.4); discount factor γ (Section 1.5.1); state-value function V, V^π (Sections 1.5, 1.11); policy π (Section 1.9); learning-rate conditions (Section 3.5).

This is the first target-construction method for unknown $V^\pi(s)$. Monte Carlo waits until the episode finishes, computes the return that actually happened, and uses that return as the sample target.

Monte Carlo evaluation estimates values from complete sampled episodes. If a state is visited at time t and the later realized return is g_t, then g_t is one sample of "how good it was to be in that state at that time".

Symbol guide.

- Episode: one observed sequence $s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T$ generated by following the fixed policy π.
- g_t: realized return computed from rewards after time t; for example $g_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$.
- $N(s)$: number of times state s has been used in the estimate.
- $G(s)$: accumulated sum of observed returns from state s.
- $\widehat{V}^\pi(s)$: current estimate of $V^\pi(s)$.

3.2.1 Algorithm flow

```
input: fixed policy pi, discount gamma
initialize: G(s)=0, N(s)=0 for all states s
repeat many times:
    run one whole episode while following pi
    for each state s visited at time t:
        compute g_t = r_t + gamma*r_{t+1} + gamma^2*r_{t+2} + ...
        N(s) = N(s) + 1
        G(s) = G(s) + g_t
        V(s) = G(s) / N(s)
output: estimated V^pi(s) for all states s
```

The important point is that Monte Carlo waits until it can see the later rewards. Only then can it compute g_t.

Incremental update form:

$$V(s) \leftarrow V(s) + \alpha\,(g_t - V(s))\,.$$

If $\alpha = 1/N(s)$, this is exactly an online average of all observed returns from state s.

Example 3.1 (Monte Carlo return from a tiny episode)**.**
Suppose an episode under policy π is

$$L \xrightarrow{r=0} C \xrightarrow{r=4} C \xrightarrow{r=4} \text{end},$$

and let $\gamma = 0.5$. The return from the first state L is

$$g_1 = 0 + 0.5 \cdot 4 + 0.5^2 \cdot 4 = 3.$$

So this episode contributes one sample saying: "starting from L produced return 3 in this episode." After many episodes, we average such samples to estimate $V^\pi(L)$.

3.2.2 First-visit and every-visit MC

- First-visit MC uses only the first occurrence of a state in an episode.
- Every-visit MC uses every occurrence of the state.

Both are based on episode returns. First-visit is cleaner statistically; every-visit can use more data from the same episode.

> **Common trap.** Monte Carlo methods usually need episodes to finish, because the return g_t is not known until later rewards are observed.

> **Takeaway. Remaining issue.** MC targets are easy to understand but inconvenient when episodes are long or do not naturally end. The next section introduces TD learning, which uses a shorter target and can update after only one transition.

3.3 Temporal-difference learning

> **Referenced concepts.** Used here: reward R / r_t (Section 1.3); discount factor γ (Section 1.5.1); policy π (Section 1.9); Monte Carlo evaluation (Section 3.2); learning-rate conditions (Section 3.5).

This section solves the problem left by Monte Carlo evaluation. TD allows the value estimate to be updated before the whole future has been observed. Temporal-difference learning, usually TD learning, also evaluates a fixed policy π. The difference from Monte Carlo is that TD updates after one observed transition instead of waiting for the whole episode.

> **Symbol guide.**
>
> - s_t: current observed state.
> - r_t: observed immediate reward after taking the policy's action in s_t.
> - s_{t+1}: observed next state.
> - $V(s_t)$: current value-table entry for the current state.
> - $V(s_{t+1})$: current value-table entry for the next state. It is not newly computed from the transition; it is whatever estimate the algorithm currently stores for s_{t+1}.
> - α: learning rate; how strongly the new target changes the old estimate.
> - TD target: $r_t + \gamma V(s_{t+1})$.
> - TD error: target minus current estimate.

$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_t + \gamma V(s_{t+1}) - V(s_t) \right).$$

The target is

$$r_t + \gamma V(s_{t+1}).$$

The TD error is

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t).$$

3.3.1 Source of $V(s_{t+1})$ in TD

It comes from the current table of estimates. At the start, we often initialize all values to zero:

$$V(s) = 0 \quad \text{for every state } s.$$

Later, as the agent observes transitions, the entries in the table are updated. When TD needs $V(s_{t+1})$, it simply looks up the current value stored for the next state.

This is called **bootstrapping**: we update one estimate using another estimate. That is why TD can update before the episode ends, but also why early TD targets can be inaccurate.

3.3.2 Mental picture of one TD step

At one moment the agent is in state s_t. It follows the fixed policy π, so the policy chooses an action. The environment then returns two observations: the immediate reward r_t and the next state s_{t+1}. TD does not know the true future after s_{t+1}. It only opens its current value table and reads the number currently stored as $V(s_{t+1})$. The update therefore says:

$$\text{new guess for current state} \approx \text{reward now} + \gamma \cdot \text{old guess for next state}.$$

So $V(s_{t+1})$ is not magic and not directly observed. It is the algorithm's current guess, reused as a training target for the previous state.

3.3.3 Algorithm flow

```
input: fixed policy pi, discount gamma, learning-rate schedule alpha
initialize: V(s)=0 for all states s
repeat for many episodes or time steps:
    observe current state s
    choose action a according to fixed policy pi
    perform action a
    observe reward r and next state s_next
    target = r + gamma * V(s_next)
    error = target - V(s)
    V(s) = V(s) + alpha * error
    s = s_next
output: estimated V^pi(s) for all states s
```

Only the value of the current state s_t is updated in that step. The value $V(s_{t+1})$ is used as a lookup in the target. It will be updated when s_{t+1} becomes the current state in a later step.

> **ELI5.** Monte Carlo waits until the end of the whole trip and uses the total points collected after the current state. TD uses the reward now plus the current estimate of the next state, so it can improve its estimate immediately.

Example 3.2 (TD as a table update)**.**
Suppose there are two states, library L and cafe C. We are evaluating a fixed policy π that generates transitions. The value table is currently

$$V(L) = 0, \qquad V(C) = 0.$$

Let $\gamma = 0.5$ and $\alpha = 0.1$.

Step 1. We observe one transition:

$$L \xrightarrow{r=1} C.$$

The TD target for L is

$$r + \gamma V(C) = 1 + 0.5 \cdot 0 = 1.$$

The update is

$$V(L) \leftarrow 0 + 0.1(1 - 0) = 0.1.$$

Now the table is

$$V(L) = 0.1, \qquad V(C) = 0.$$

Step 2. Later we observe:

$$C \xrightarrow{r=4} C.$$

The TD target for C is

$$r + \gamma V(C) = 4 + 0.5 \cdot 0 = 4.$$

So

$$V(C) \leftarrow 0 + 0.1(4 - 0) = 0.4.$$

Now the table is

$$V(L) = 0.1, \qquad V(C) = 0.4.$$

Step 3. If we again observe

$$L \xrightarrow{r=1} C,$$

then $V(C)$ is no longer zero. The target for L is now

$$1 + 0.5 \cdot 0.4 = 1.2.$$

So

$$V(L) \leftarrow 0.1 + 0.1(1.2 - 0.1) = 0.21.$$

This is the flow: TD slowly propagates value information backward from later states to earlier states.

> **Takeaway. Where we are now.** MC and TD are two ways to build targets for estimating values under a fixed policy. The next section compares them directly, then we discuss learning rates, and only after that we move to control algorithms that improve behaviour.

3.4 Comparison: Monte Carlo versus TD

This comparison belongs to policy evaluation: both methods estimate V^π for a fixed policy. The difference is not the final object, but how the training target is obtained.

> **Referenced concepts.** Used here: policy π (Section 1.9); Monte Carlo evaluation (Section 3.2); features (Section 5.2).

Feature	Monte Carlo	Temporal Difference
What it evaluates	fixed policy π	fixed policy π
Creates policy	no	no
Needs complete episode	usually yes	no
Uses model P, R	no	no
Uses bootstrapping	no	yes, uses current $V(s_{t+1})$
Variance	often higher	often lower
Bias	can be unbiased in first-visit form	biased during learning due to bootstrapping

3.5 Learning-rate convergence conditions

> **Referenced concepts.** Used here: state-value function V, V^π (Sections 1.5, 1.11); action-value function Q^π (Section 1.12); DQN (Section 5.8); TD learning (Section 3.3); function approximation (Section 5.1).

For stochastic approximation style TD learning, the learning rates α_t should usually satisfy

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \qquad \sum_{t=1}^{\infty} \alpha_t^2 < \infty.$$

- $\sum \alpha_t = \infty$ means the algorithm never completely stops learning too early.
- $\sum \alpha_t^2 < \infty$ means the steps become small enough that random noise does not shake the estimates forever.

Examples that satisfy both conditions:

$$\alpha_t = \frac{1}{t}, \qquad \alpha_t = \frac{1}{t^{0.6}}, \qquad \alpha_t = \frac{1}{N_t(s)} \text{ for state-value updates of state } s.$$

For action-value learning, the analogous safe example is

$$\alpha_t(s,a) = \frac{1}{N_t(s,a)},$$

where $N_t(s,a)$ is the number of times the pair (s,a) has already been updated. More generally, $\alpha_t = 1/t^p$ satisfies both when

$$0.5 < p \leq 1.$$

Examples that do not satisfy both:

- Constant $\alpha_t = 0.1$: keeps adapting, but $\sum \alpha_t^2 = \infty$, so it does not settle in the strict convergence-theorem sense.
- $\alpha_t = 1/t^2$: steps become too small too fast, because $\sum \alpha_t < \infty$.

- $\alpha_t = 1/\sqrt{t}$: keeps learning, but $\sum \alpha_t^2 = \sum 1/t = \infty$.

> **Exam use.** The model exam asks about these two conditions. The short answer is: infinite sum of learning rates, finite sum of squared learning rates. A safe example is $\alpha_t = 1/t$ or, for a particular state, $\alpha_t(s) = 1/N_t(s)$.

3.6 Bridge to control: from $V(s)$ to $Q(s, a)$

So far the chapter estimated how good states are under a fixed policy. That is evaluation. To improve behaviour, the agent must compare actions inside a state. This is why control algorithms usually learn $Q(s, a)$ instead of only $V(s)$: $Q(s, a)$ tells us how good it is to take action a in state s.

> **Referenced concepts.** Used here: policy π (Section 1.9); action-value function Q^π and table estimate $Q(s, a)$ (Section 1.12); TD learning (Section 3.3); evaluation vs control (Section 4.1); SARSA (Section 4.5); Q-learning (Section 4.6).

The same confusion can appear in SARSA and Q-learning (Sections 4.5 and 4.6). There, the table stores $Q(s, a)$ instead of $V(s)$. This is why the next chapter mostly stops talking about $V(s)$ and starts talking about $Q(s, a)$: for control, we need to compare actions, not only states.

- In TD evaluation, the table entry is $V(s)$: value of a state under a fixed policy.
- In SARSA/Q-learning, the table entry is $Q(s, a)$: value of taking action a in state s.
- In both cases, the "next value" in the target is usually just a lookup in the current table.

Chapter 4

Lecture 3: Model-Free Control

This chapter continues directly from model-free evaluation. Evaluation estimated values for a fixed policy. Control tries to improve the policy. The central switch is from state values $V(s)$ to action values $Q(s, a)$, because choosing actions requires comparing actions.

> **Takeaway. Roadmap of this chapter.** First we separate evaluation from control. Then we introduce exploration and the on-policy/off-policy distinction. After that, SARSA and Q-learning become much easier to understand: SARSA updates from the action actually selected next, while Q-learning updates from the best currently estimated next action.

4.1 From evaluation to control

> **Referenced concepts.** Used here: state-value function V, V^π (Sections 1.5, 1.11); MDP (Section 1.7); policy π (Section 1.9); action-value function Q^π (Section 1.12); policy evaluation (Section 2.1); ε-greedy (Section 4.2).

Policy evaluation estimates how good a fixed policy is. Control searches for a better or optimal policy.

In model-free control, we do not know the MDP parameters. Therefore we learn action-values $Q(s, a)$ from interaction and improve behavior from those estimates. Here $Q(s, a)$ means the learner's table estimate of the action-value function introduced in Section 1.12.

> **Symbol guide.**
>
> - $Q(s, a)$ without superscript usually means the learner's current estimate, not necessarily the exact Q^π.
> - $Q^*(s, a)$ is the optimal action-value function.
> - Control output: a policy, usually obtained by acting greedily or ε-greedily with respect to learned Q.

> **ELI5.** Evaluation grades a strategy that is already fixed. Control learns a better strategy. That is why evaluation outputs values for a given policy, while control eventually outputs or implies a policy.

4.2 Exploration and exploitation

Referenced concepts. Used here: policy π (Section 1.9); evaluation vs control (Section 4.1).

A control agent faces a conflict:

- exploitation: use the currently best-looking action;
- exploration: try other actions to learn whether they are better.

Definition 4.1 (ε-greedy policy)**.**
Given $Q(s,a)$ (the current action-value estimate; Section 1.12), an ε-greedy policy chooses a greedy action with high probability and a random action with small probability. For one greedy action a^*:

$$\pi(a \mid s) = \begin{cases} 1-\varepsilon+\varepsilon/|A|, & a = a^*, \\ \varepsilon/|A|, & a \neq a^*. \end{cases}$$

4.3 On-policy versus off-policy

This terminology is placed before SARSA and Q-learning because it explains their main difference. It is not a separate trick; it tells us whether the algorithm learns the value of the behaviour policy itself or of some other target policy.

Referenced concepts. Used here: policy π (Section 1.9); action-value function Q^π and table estimate $Q(s,a)$ (Section 1.12); evaluation vs control (Section 4.1); exploration/exploitation and ε-greedy (Section 4.2). This section is placed before SARSA and Q-learning because those algorithms only make sense after the behavior/target-policy distinction is clear.

In control algorithms there are often two policies to keep apart:

- **behavior policy**: the policy that actually chooses actions during learning, for example an ε-greedy exploratory policy;
- **target policy**: the policy whose values the algorithm is trying to learn.

Algorithm type	Meaning	Example
On-policy	behavior policy and target policy are the same; the algorithm evaluates/improves the way it actually behaves	SARSA
Off-policy	behavior policy and target policy may differ; the algorithm may behave exploratorily while learning values for a greedier target policy	Q-learning

Example 4.1 (Same observed transition, different target)**.**
Suppose both algorithms observe the same current transition and land in next state C. The behavior policy then randomly chooses `wander`, even though the best estimated action in C is `study`:

$$Q(C,\text{study}) = 8, \qquad Q(C,\text{wander}) = 2.$$

SARSA target uses the action actually selected by behavior:

$$r + \gamma Q(C,\text{wander}) = r + 2\gamma.$$

Q-learning target uses the best estimated next action:

$$r + \gamma \max_a Q(C, a) = r + 8\gamma.$$

This is the core difference between on-policy and off-policy in this part of the course.

ELI5. On-policy means: learn the value of the strategy you are actually using, including its random exploration. Off-policy means: you may behave randomly for exploration, but you update as if you want to learn a different, usually greedy, strategy.

4.4 Monte Carlo control

Monte Carlo control is the most direct extension of Monte Carlo evaluation. Instead of only estimating $V^\pi(s)$ for a fixed policy, we estimate $Q(s, a)$ from completed episodes and then make the policy more greedy with respect to those estimates.

Referenced concepts. Used here: policy π (Section 1.9); Monte Carlo evaluation (Section 3.2); evaluation vs control (Section 4.1); exploration/exploitation (Section 4.2); ε-greedy (Section 4.2).

Monte Carlo control estimates $Q(s, a)$ (state-action values; Section 1.12) from complete episodes and then improves the policy greedily or ε-greedily.

A naive greedy update can fail because some actions may never be tried. Therefore exploration must be built into the behavior policy.

4.5 SARSA

Referenced concepts. Used here: reward R / r_t (Section 1.3); discount factor γ (Section 1.5.1); policy π (Section 1.9); policy evaluation (Section 2.1); TD learning (Section 3.3); learning-rate conditions (Section 3.5); evaluation vs control (Section 4.1); action-value function Q^π and table estimate $Q(s, a)$ (Section 1.12).

SARSA is an on-policy TD control algorithm. Its name comes from the tuple

$$(s_t, a_t, r_t, s_{t+1}, a_{t+1}).$$

Unlike TD policy evaluation, SARSA is a control algorithm: it learns $Q(s, a)$ (the current state-action value estimate; Section 1.12) and can derive a policy from it, usually by using an ε-greedy policy.

Symbol guide.

- $Q(s, a)$: current table estimate for taking action a in state s.
- a_{t+1}: the next action actually selected by the current behavior policy.
- SARSA target: $r_t + \gamma Q(s_{t+1}, a_{t+1})$.

The update is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right).$$

4.5.1 Algorithm flow

```
initialize Q(s,a)=0
choose first action a using epsilon-greedy(Q)
repeat:
    perform action a in state s
    observe reward r and next state s_next
    choose next action a_next using epsilon-greedy(Q)
    target = r + gamma * Q(s_next, a_next)
    Q(s,a) = Q(s,a) + alpha * (target - Q(s,a))
    s = s_next
    a = a_next
```

4.5.2 Randomness in SARSA

The randomness is not written as a separate random variable inside the final formula. It enters through the sampled next action a_{t+1}. In SARSA, a_{t+1} is chosen by the same behavior policy that the agent actually uses, often an ε-greedy policy. Therefore sometimes a_{t+1} is the greedy action and sometimes it is an exploratory random action.

For example, suppose that after one transition the next state is $s_{t+1} = C$ and the current table says

$$Q(C, \text{study}) = 8, \qquad Q(C, \text{wander}) = 2.$$

If the ε-greedy policy actually samples $a_{t+1} = \text{wander}$, then SARSA uses

$$r_t + \gamma Q(C, \text{wander})$$

as the target, not $r_t + \gamma Q(C, \text{study})$. This is why SARSA learns the value of the behavior policy including exploration. The sampled action appears directly in $Q(s_{t+1}, a_{t+1})$.

> **ELI5.** SARSA is like learning from the path you really took. If you sometimes make random exploratory moves, those moves affect the next action a_{t+1}, and that exact sampled action is used in the update target.

4.6 Q-learning

Q-learning is the off-policy TD-control counterpart to SARSA. It still learns from sampled transitions, but its update target assumes greedy behaviour from the next state onward. This section therefore completes the sequence: MC control waits for full returns, SARSA uses the actually sampled next action, and Q-learning uses the best currently estimated next action.

> **Referenced concepts.** Used here: reward R / r_t (Section 1.3); discount factor γ (Section 1.5.1); policy π (Section 1.9); learning-rate conditions (Section 3.5); ε-greedy (Section 4.2); SARSA (Section 4.5); on-policy/off-policy (Section 4.3); action-value function Q^π and table estimate $Q(s, a)$ (Section 1.12).

Q-learning is off-policy. It also learns a $Q(s, a)$ table (state-action value estimates; Section 1.12), but it differs from SARSA in the action used for the next-state part of the update.

After the agent takes action a_t in state s_t, it observes reward r_t and lands in the next state s_{t+1}. If the behavior policy is ε-greedy, the next action in s_{t+1} may be greedy or it may be a random exploratory action.

SARSA uses the action that the agent actually chooses next in this sampled episode. That sampled action a_{t+1} appears in $Q(s_{t+1}, a_{t+1})$.

Q-learning uses the best action currently estimated for the next state. This is the term $\max_{a'} Q(s_{t+1}, a')$, even if the behavior policy actually sampled some other exploratory action.

Symbol guide.

- $Q(s, a)$: current table estimate for state-action pair (s, a).
- $\max_{a'} Q(s_{t+1}, a')$: best currently estimated action value in the next state.
- Q-learning target: $r_t + \gamma \max_{a'} Q(s_{t+1}, a')$.

The update is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right).$$

It learns about the greedy target policy even if the behavior policy explores.

Compare this with SARSA. If the next state is C and the current table has $Q(C, \text{study}) = 8$ and $Q(C, \text{wander}) = 2$, and the ε-greedy behavior policy actually sampled the exploratory action `wander`, then SARSA would use $Q(C, \text{wander}) = 2$. Q-learning instead uses $\max(8, 2) = 8$. That is the practical meaning of off-policy here.

4.6.1 Algorithm flow

```
initialize Q(s,a)=0
repeat:
    choose action a using behavior policy, often epsilon-greedy(Q)
    perform action a in state s
    observe reward r and next state s_next
    target = r + gamma * max_a Q(s_next, a)
    Q(s,a) = Q(s,a) + alpha * (target - Q(s,a))
    s = s_next
```

ELI5. Q-learning says: I may have chosen actions randomly to collect experience, but the update is about the greedy policy I am trying to learn. SARSA learns from the next action that was actually sampled; Q-learning learns from the best currently estimated next action.

Example 4.2 (SARSA versus Q-learning target)**.**
Suppose $r = 1$, $\gamma = 0.9$, $Q(s, a) = 2$, and in the next state

$$Q(s', \text{left}) = 4, \quad Q(s', \text{right}) = 10.$$

If the next action actually sampled is left, SARSA target is

$$1 + 0.9 \cdot 4 = 4.6.$$

Q-learning target is

$$1 + 0.9 \cdot 10 = 10.$$

Same transition, different learning target.

4.7 Important adversarial-action intuition

Referenced concepts. Used here: exploration/exploitation (Section 4.2); SARSA (Section 4.5); Q-learning (Section 4.6).

The course sometimes phrases the comparison using an adversarial or corrupted behaviour policy. The safe interpretation is this: the learner still observes a valid transition tuple, but the behaviour policy that produced experience may choose actions that are not greedy.

Under that interpretation:

- SARSA uses the actually followed next action in its target, so it learns values for the behaviour including such disturbances.
- Q-learning uses $\max_a Q(s', a)$ in the next-state part of the target, so it can still learn optimal greedy values if the usual convergence assumptions and sufficient exploration are satisfied.

Common trap. Do not overgeneralize this. If the logged action a_t in the update is not the action that actually caused the observed reward and next state, then the data are inconsistent with the update rule and the standard guarantee does not apply. The exam-level point is the on-policy/off-policy difference in the target, not that Q-learning magically survives arbitrary corrupted data.

Exam use. For questions comparing SARSA and Q-learning, always mention the target: SARSA target uses the next sampled action; Q-learning target uses a max over next actions.

4.8 GLIE conditions

Referenced concepts. Used here: policy π (Section 1.9); exploration/exploitation (Section 4.2); ε-greedy (Section 4.2).

GLIE means "greedy in the limit with infinite exploration". Informally:

- every state-action pair is explored infinitely often;
- the policy becomes greedy in the limit.

This is why schedules such as decreasing ε are used. Constant ε explores forever but never becomes fully greedy. Too-fast decay may become greedy before enough exploration.

Common trap. Do not say that random exploration alone guarantees optimal play. It may explore forever, but it may never become greedy.

Chapter 5

Lecture 4: Function Approximation

The previous chapters assumed a table: one value for every state, or one value for every state-action pair. Function approximation replaces this table by a parameterized function. This chapter is one continuous story: tables are too large, so we use features and approximators; approximators need training targets; MC and TD provide those targets; control then uses approximate Q values.

> **Takeaway. Roadmap of this chapter.** First we replace tables by functions. Then we explain the supervised-learning view. The key missing object is the target y. Monte Carlo and TD are not re-taught here as independent algorithms; they are reused briefly as two target choices for function approximation.

5.1 Why tabular methods are limited

> **Referenced concepts.** Used here: state-value function V and V^π (Sections 1.5, 1.11); action-value function Q^π (Section 1.12); return G_t (Section 1.4); features (Section 5.2).

Earlier algorithms store one number for every state or every state-action pair. This is called a **tabular** representation because the learned object is literally a table:

state s	$V(s)$
s_1	3.2
s_2	7.8

or

state s	action a	$Q(s,a)$
s_1	a_1	2.1
s_1	a_2	5.4

This works only when the state space is small enough to store and visit many times. For image input, continuous physical states, or many symbolic combinations, most states may be rare or unique. Then we need to **generalize**: if two states look similar, their values should usually be similar too.

Definition 5.1 (Function approximation)**.**
Instead of storing an exact table entry for $V(s)$ or $Q(s,a)$, we represent the value by a parameterized function:

$$\widehat{V}(s;w) \quad \text{or} \quad \widehat{Q}(s,a;w),$$

where w are learned parameters.

> **Symbol guide.**

- $V(s)$: exact/ideal state-value function, one real number for state s (see Sections 1.5 and 1.11).
- $Q(s, a)$: action-value function, one real number for taking action a in state s (see Section 1.12).
- The hat in $\widehat{V}$ or $\widehat{Q}$ means **estimate/approximation**. It says: "this is what our model currently predicts", not the true value.
- $\widehat{V}(s; w)$: approximate value of state s using parameters w.
- $\widehat{Q}(s, a; w)$: approximate value of state-action pair (s, a) using parameters w.
- w: parameter vector, e.g. linear weights or neural-network weights. These are the numbers we actually learn.
- $x(s)$: feature vector representation of state s (defined in Section 5.2).
- Output type: still a real number estimating expected discounted return, not a state, not an action, and not an immediate reward.

ELI5. A table is like a dictionary where every state needs its own entry. Function approximation is like learning a rule. Instead of remembering every possible position separately, the agent learns: "positions with these features are usually good" or "positions that look like this are dangerous". The hat $\widehat{V}$ means the rule's current guess of the value.

Takeaway. Transition. A function cannot work directly with a vague "state". We first encode the state as numbers. Those numbers are called features.

5.2 Features

Referenced concepts. Used here: state s (Section 1.1); function approximation (Section 5.1).

A function approximator usually does not work directly with a symbolic state name. It first converts the state into numbers. A state is represented by a feature vector

$$x(s) = (x_1(s), \ldots, x_d(s))^T.$$

Symbol guide.

- s: the current state, e.g. a screen image, robot position, or symbolic situation.
- $x(s)$: vector of numbers describing state s.
- $x_i(s)$: the i-th feature of state s.
- d: number of features.
- T: transpose. The notation $(\cdots)^T$ says that we treat the list as a column vector.

Examples:

- pixels of an Atari screen;
- position, velocity, and angle in pole balancing;
- symbolic indicators in a discrete problem, e.g. "has key", "door open", "enemy nearby".

ELI5. A feature vector is a checklist written as numbers. Instead of saying "this exact state is number 981242", we describe it: position, speed, angle, whether something is nearby, and so on. The approximator can then learn that similar checklists should often have similar values.

Takeaway. Transition. Once states are feature vectors, the simplest approximator is a weighted sum of these features.

5.3 Linear value approximation

Referenced concepts. Used here: state-value function V, V^π (Sections 1.5, 1.11); function approximation (Section 5.1); features (Section 5.2).

A simple approximation is

$$\widehat{V}(s; w) = w^T x(s).$$

Read this as: multiply every feature by its learned weight, then add the results. This gives one number: the predicted value of state s.

Symbol guide.

- $\widehat{V}(s; w)$: predicted/approximate state value of state s.
- $w = (w_1, \ldots, w_d)^T$: learned weights, one weight per feature.
- $x(s) = (x_1(s), \ldots, x_d(s))^T$: feature vector of state s.
- $w^T x(s)$: dot product, i.e. $w_1 x_1(s) + \cdots + w_d x_d(s)$.

This can work well when the features are good. It is interpretable and easy to update, but it depends heavily on feature engineering.

Example 5.1 (Linear value estimate)**.**
Suppose a state has two features:

$$x(s) = \begin{bmatrix} 1 \\ 3 \end{bmatrix},$$

where the first feature might mean "has key" and the second feature might mean "distance from goal". Suppose the learned weights are

$$w = \begin{bmatrix} 5 \\ -1 \end{bmatrix}.$$

Then

$$\widehat{V}(s; w) = w^T x(s) = 5 \cdot 1 + (-1) \cdot 3 = 2.$$

So the current model estimates the value of this state as 2.

Takeaway. Transition. Linear approximation is simple and interpretable. Neural approximation keeps the same input-output story but uses a more flexible function.

5.4 Neural approximation

Referenced concepts. Used here: state-value function V, V^π (Sections 1.5, 1.11); features (Section 5.2); gradient descent (Section 5.5).

A neural network approximation can be written abstractly as

$$\widehat{V}(s; w) = g(x(s); w).$$

Here g is the neural network. The lecture treats neural networks as differentiable black boxes: we can evaluate them and compute gradients.

Symbol guide.

- g: the prediction function implemented by the neural network.
- $x(s)$: input features of state s.
- w: all trainable parameters of the network, e.g. weights and biases.
- $\widehat{V}(s; w)$: the network output, interpreted as approximate value of state s.
- "Differentiable" means we can compute how changing w changes the output/loss, which is needed by gradient descent.

ELI5. Linear approximation is like using a simple weighted checklist. A neural network is a more flexible rule. It still receives numbers describing the state and outputs one number: its guess of the value.

Takeaway. Transition. Whether the approximator is linear or neural, it has parameters w. We need a way to adjust those parameters. That is the role of gradient descent.

5.5 Gradient descent and SGD

Referenced concepts. Used here: learning-rate conditions (Section 3.5); function approximation (Section 5.1).

Gradient descent updates parameters in the direction that decreases a loss:

$$w_{k+1} = w_k - \alpha \nabla J(w_k).$$

Stochastic gradient descent, or SGD, estimates the gradient using one sample or a mini-batch instead of the whole expected loss.

Symbol guide.

- w_k: current parameter vector at optimization step k.
- w_{k+1}: next parameter vector after one update.
- $J(w)$: loss/objective function. Smaller is better.
- $\nabla J(w_k)$: gradient of the loss at w_k, i.e. direction of steepest increase.
- α: learning rate / step size. Larger α changes parameters more aggressively.

- k: optimization-step index, not an RL time step.

ELI5. Imagine being on a hill in fog. The gradient tells you which direction goes uphill fastest. To minimize, you step the other way. SGD estimates that direction from a small noisy clue instead of seeing the whole hill.

Takeaway. Transition. Gradient descent needs a loss, and the loss needs a target. The next section explains value approximation as supervised learning: the approximator predicts a value and is trained toward a numeric target.

5.6 Value approximation as supervised learning

Referenced concepts. Used here: state-value function V, V^π (Sections 1.5, 1.11); Monte Carlo evaluation (Section 3.2); TD learning (Section 3.3); function approximation (Section 5.1); gradient descent (Section 5.5).

This section connects the previous ideas instead of starting a new topic. We already know how tables are evaluated: MC uses full returns, TD uses one-step bootstrapping. Function approximation keeps the same learning logic, but replaces the table entry $V(s)$ by a prediction $\widehat{V}(s; w)$.

The approximator can output $\widehat{V}(s; w)$, but training requires a target value. Therefore the missing object is the target:

$$\text{target value for } \widehat{V}(s; w)$$

If we had target values y_s for states, we could minimize mean squared error:

$$J(w) = \mathbb{E}[(y_s - \widehat{V}(s; w))^2].$$

This says: adjust parameters w so that the prediction $\widehat{V}(s; w)$ is close to the target y_s.

Symbol guide.

- s: state used as a training input.
- y_s: temporary target value for state s in this update; it is the number we want $\widehat{V}(s; w)$ to move toward.
- $\widehat{V}(s; w)$: current prediction of the function approximator.
- w: parameters of the approximator, for example linear weights or neural-network weights.
- $J(w)$: average squared prediction error.
- $\mathbb{E}[\cdot]$: expectation/average over sampled states or transitions.

In ordinary supervised learning, y_s would be given in the dataset. In RL, the correct answer is usually not known directly. If we already knew the true value $V^\pi(s)$ for every state, then learning $\widehat{V}$ would only be compression or approximation, not real policy evaluation.

Origin of y_s in RL. The symbol y_s is not hidden internal knowledge. It is a temporary training target produced from observed experience. There are two main choices, both already introduced in the model-free evaluation chapter:

- MC target: use the full observed return G_t after the episode ends.
- TD target: use one observed reward plus the current prediction of the next state.

> **ELI5.** The function approximator is like a student guessing values. The target y_s is the answer we want it to move toward. In normal supervised learning the teacher gives the answer. In RL, the algorithm makes a temporary answer from experience: MC uses the score actually collected later, while TD uses reward now plus the current guess for the next state.

5.6.1 MC target for value approximation

> **Referenced concepts.** Used here: return G_t (Section 1.4); Monte Carlo evaluation (Section 3.2); state-value function V^π (Section 1.11); function approximation (Section 5.1).

This is not a separate new MC algorithm. It is the same MC idea from Section 3.2, used as a target generator for an approximator.

After an episode ends, compute the realized return from time t:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots .$$

For the visited state s_t, set the supervised-learning target to

$$y_{s_t} = G_t.$$

Then update w so that $\widehat{V}(s_t; w)$ becomes closer to G_t.

> **Symbol guide.**
>
> - s_t: state visited at time t.
> - G_t or g_t: realized return from time t in this particular episode.
> - $\widehat{V}(s_t; w)$: current approximate prediction for state s_t.
> - Target for this update: $y_{s_t} = G_t$.

> **ELI5.** MC approximation waits until the episode is over. Then it can say: "When I was in this state, the later collected score was G_t. I will train the value function to predict something closer to that."

5.6.2 TD target for value approximation

> **Referenced concepts.** Used here: discount factor γ (Section 1.5.1); TD learning (Section 3.3); bootstrapping (Section 3.3); state-value function V^π (Section 1.11); function approximation (Section 5.1).

This is the same TD idea from Section 3.3, used as a target generator for an approximator. It does not wait for the full return. After one observed transition

$$s_t \xrightarrow{r_t} s_{t+1},$$

it uses

$$y_t = r_t + \gamma \widehat{V}(s_{t+1}; w).$$

Then it reduces the error between the current prediction $\widehat{V}(s_t; w)$ and the target y_t.

Symbol guide.

- s_t: current state.
- r_t: reward observed after the transition from s_t.
- s_{t+1}: next state.
- $\widehat{V}(s_{t+1}; w)$: current approximate prediction for the next state.
- y_t: TD target; the number we want $\widehat{V}(s_t; w)$ to move toward.

The logic is exactly the same as tabular TD, but table lookup $V(s)$ is replaced by function prediction $\widehat{V}(s; w)$:

$$\text{tabular TD: } r_t + \gamma V(s_{t+1}) \qquad \longrightarrow \qquad \text{approximate TD: } r_t + \gamma \widehat{V}(s_{t+1}; w).$$

Common trap. With function approximation, changing parameters for one state can change predictions for many other states. This is useful for generalization but can also destabilize learning. In a table, updating $V(s_t)$ changes only one table cell. With $\widehat{V}(s; w)$, updating w may change predictions for many states.

Takeaway. Transition. So far we approximated state values for evaluation. For control, we approximate action values, because choosing actions requires comparing actions.

5.7 Control with function approximation

Referenced concepts. Used here: action-value function Q^π and $Q(s, a)$ (Section 1.12); Monte Carlo evaluation (Section 3.2); TD learning (Section 3.3); on-policy/off-policy (Section 4.3); SARSA (Section 4.5); Q-learning (Section 4.6); function approximation (Section 5.1); features (Section 5.2).

For control, we usually approximate action-values rather than only state-values:

$$Q(s, a) \approx \widehat{Q}(s, a; w).$$

Then SARSA or Q-learning can be written with approximate values. This is the conceptual step toward deep reinforcement learning.

Symbol guide.

- $Q(s, a)$: value of taking action a in state s.
- $\widehat{Q}(s, a; w)$: approximator's current prediction of that value.

- Input to $\widehat{Q}$: state s, action a, and parameters w.
- Output of $\widehat{Q}$: one real number estimating expected discounted return.

ELI5. State-value approximation estimates how good a state is. Action-value approximation estimates how good an action is in a state. Control needs action values because choosing an action requires comparing actions.

Exam use. You probably do not need deep derivations here. Know why approximation is needed, what features are, what the hat means in $\widehat{V}(s;w)$ and $\widehat{Q}(s,a;w)$, and how MC/TD targets become supervised-learning targets.

5.8 Deep Q-learning / DQN

Referenced concepts. Used here: Q-learning (Section 4.6); action-value approximation (Section 5.7); neural approximation (Section 5.4); ε-greedy (Section 4.2).

Deep Q-learning, usually called DQN, is the natural continuation of Q-learning with function approximation. The Q-table is replaced by a neural network:

$$Q(s,a) \approx \widehat{Q}(s,a;w).$$

This is needed when the state is too large for a table, for example Atari frames represented by pixels.

Symbol guide.

- w: current neural-network parameters.
- w^-: target-network parameters, copied from w only occasionally.
- D: replay buffer storing previous transitions (s,a,r,s').
- C: number of steps between target-network updates.
- y_i: DQN target for one sampled transition.

For one transition (s_i,a_i,r_i,s_{i+1}), the DQN target is

$$y_i = \begin{cases} r_i, & \text{if } s_{i+1} \text{ is terminal,} \\ r_i + \gamma \max_{a'} \widehat{Q}(s_{i+1},a';w^-), & \text{otherwise.} \end{cases}$$

Then the network is trained to reduce

$$\Big(y_i - \widehat{Q}(s_i,a_i;w)\Big)^2.$$

Reason DQN needs extra tricks. Plain neural Q-learning can be unstable. The two standard problems are not mysterious; they come from using a neural network inside a bootstrapping update.

- **Consecutive samples are correlated.** If the agent is walking through one trajectory, then the next few transitions are usually very similar. For example, in an Atari game, consecutive frames often show almost the same screen. Training only on such nearby samples is like repeatedly training on almost the same kind of example. The network can overreact to the most recent local situation and forget or damage predictions for other states.
- **The target is moving.** In ordinary supervised learning the label is fixed. If an image is labelled "cat", the label does not change during training. In Q-learning the target contains another prediction of the network:

  $$r + \gamma \max_{a'} \widehat{Q}(s', a'; w).$$

  So when w changes, both the prediction being trained and the target can change. This is like chasing a label that keeps moving while you train.

DQN therefore uses two stabilizers:

- **experience replay**: store transitions in D and train on random minibatches sampled from many older transitions;
- **target network**: compute the target with w^-, which is held fixed for several steps and only sometimes copied from w.

Stabilization from experience replay. The replay buffer does not make the environment itself uncorrelated. The original trajectory is still correlated. The change is in the *training batch*. Instead of updating the network from the last few transitions in order,

$$(s_1, a_1, r_1, s_2), (s_2, a_2, r_2, s_3), (s_3, a_3, r_3, s_4), \ldots$$

DQN stores many transitions and samples a random minibatch from the buffer. One minibatch can contain old transitions from many different parts of many episodes. This makes the gradient update less dominated by the most recent local trajectory.

This helps in three ways:

1. **Less correlated updates.** The examples in one minibatch are more mixed, so the neural network is less likely to chase one short sequence of almost identical states.
2. **Better data efficiency.** One transition can be reused many times. Without replay, the agent observes a transition once and then immediately throws it away.
3. **Less forgetting.** Because old transitions keep reappearing in minibatches, the network is reminded of older states and actions, not only the newest part of the trajectory.

Stabilization from the target network. Experience replay mixes the input data, but it does not fully solve the moving-target problem. The target network addresses that second problem. The update target is computed using w^-:

$$y_i = r_i + \gamma \max_{a'} \widehat{Q}(s_{i+1}, a'; w^-).$$

During several gradient steps, w changes, but w^- stays fixed. Therefore the targets are more stable for a while. After C steps, w^- is updated by copying the current online parameters w.

ELI5. Replay buffer stabilizes the examples. Target network stabilizes the labels. Replay says: "do not learn only from the last few very similar moments." Target network says: "do not change the answer key after every tiny weight update."

Common trap. Experience replay does not make DQN mathematically guaranteed to converge. It only reduces an important source of instability. Neural Q-learning can still be unstable because the function approximator generalizes, the targets are bootstrapped, and the policy changes during learning.

```
initialize Q-network weights w
initialize target weights w_minus = w
initialize replay buffer D = empty
repeat:
    choose action a using epsilon-greedy with Q_hat(s, a; w)
    perform action a, observe reward r and next state s_next
    store (s, a, r, s_next) in D
    sample random minibatch from D
    for each sampled transition (s_i, a_i, r_i, s_i_next):
        if s_i_next is terminal:
            y_i = r_i
        else:
            y_i = r_i + gamma * max_a Q_hat(s_i_next, a; w_minus)
        update w to reduce (y_i - Q_hat(s_i, a_i; w))^2
    every C steps: w_minus = w
```

ELI5. DQN is still Q-learning. The difference is storage. Instead of a table cell $Q(s, a)$, a neural network predicts $\widehat{Q}(s, a; w)$. Because the neural network can be unstable, DQN remembers old transitions in a replay buffer and uses a slower-moving target network.

Exam use. For exam/lab-level understanding, know the three names: Q-network, replay buffer, target network. Also know the stabilizing intuition: replay mixes old transitions into the training batch and reuses them; the target network makes the bootstrapped target less non-stationary.

Chapter 6

Lectures 5 and 6: Multi-Armed Bandits

Bandits continue the exploration-exploitation story but remove almost everything else. There is no long chain of future states. There is only repeated action choice and immediate reward. This makes bandits simpler than MDPs, but it makes the exploration problem very clear.

6.1 Bandits as a simplified RL problem

Referenced concepts. Used here: reward R / r_t (Section 1.3); MDP (Section 1.7); bandit action value / regret (Section 6.2).

A multi-armed bandit is like an MDP with a single state. At each time step, choose an action and receive a reward sampled from that action's reward distribution.

Definition 6.1 (Multi-armed bandit)**.**
A bandit problem consists of actions $A = \{a_1, \ldots, a_m\}$ and reward distributions $\mathbb{P}(R_t = r \mid A_t = a)$.

Symbol guide.

- There is only one state, so we do not write s.
- A_t: random action chosen at time t.
- R_t: random reward observed after choosing A_t.
- $Q(a) = \mathbb{E}[R_t \mid A_t = a]$: expected reward of action/arm a.

ELI5. You have several slot machines. Each machine pays differently, but you do not know which is best. You must earn rewards while learning which machine is good.

6.2 Action values and regret

Referenced concepts. Used here: reward R / r_t (Section 1.3).

The action value is the expected reward:

$$Q(a) = \mathbb{E}[R_t \mid A_t = a].$$

The optimal value and optimal action are

$$V^* = \max_a Q(a), \qquad a^* \in \operatorname{argmax}_a Q(a).$$

The instantaneous regret of choosing A_t is

$$L_t = V^* - Q(A_t).$$

Total regret is

$$L_T^{tot} = \sum_{t=1}^{T} L_t.$$

> **Takeaway.** Maximizing expected reward is equivalent to minimizing regret. Slow-growing regret means the algorithm is learning efficiently.

Example 6.1 (Regret)**.**
Two ads have click probabilities $Q(a_1) = 0.8$ and $Q(a_2) = 0.5$. Here $Q(a)$ means expected immediate reward of bandit arm/action a; it is not the same object as the MDP $Q^\pi(s, a)$, although the intuition is similar. The best possible expected reward per round is

$$V^* = \max_a Q(a) = 0.8.$$

If an algorithm chooses a_2 twice and a_1 eight times in ten rounds, total regret is

$$2 \cdot (0.8 - 0.5) + 8 \cdot (0.8 - 0.8) = 0.6.$$

The two choices of a_2 are the only times where the algorithm loses expected reward compared with the best arm.

6.3 Why pure greedy can fail

> **Referenced concepts.** Used here: bandit action value / regret (Section 6.2).

A greedy algorithm estimates $Q(a)$ and always chooses the action with the highest estimate. If early samples are unlucky, a good action may look bad and never be tried again. Then regret grows linearly.

> **Common trap.** Greedy is not the same as optimal learning. Greedy can stop exploring too early.

6.4 ε-greedy bandits

> **Referenced concepts.** Used here: ε-greedy (Section 4.2); bandit action value / regret (Section 6.2).

With probability $1 - \varepsilon$, choose the current best action. With probability ε, choose randomly. This prevents total early lock-in, but constant ε still causes linear regret because the algorithm keeps choosing random suboptimal actions forever.

6.5 Optimism under uncertainty and UCB

Referenced concepts. Used here: reward R / r_t (Section 1.3); exploration/exploitation (Section 4.2).

Upper Confidence Bound methods choose actions using both estimated reward and uncertainty:

$$U_t(a) = \widehat{Q}(a) + \sqrt{\frac{\log t}{2N(a)}}$$

(up to constants depending on the exact theorem). The selected action is

$$a_t \in \operatorname{argmax}_a U_t(a).$$

The second term is large for actions tried only a few times. This encourages exploration exactly where uncertainty is high.

ELI5. If two toys look equally fun but one was tried only once, UCB says: maybe the barely tried one is secretly great, so try it again.

6.6 Bayesian bandits and Beta-Bernoulli updating

Referenced concepts. Used here: learning-rate conditions (Section 3.5).

For binary rewards, such as click/no-click, each arm can be modeled as Bernoulli with unknown success probability θ.

A Beta prior is often used:

$$\theta \sim \text{Beta}(\alpha, \beta).$$

After observing p successes and n failures, the posterior is

$$\theta \mid data \sim \text{Beta}(\alpha + p, \beta + n).$$

Example 6.2 (Ad clicks)**.**
Start with $\text{Beta}(1, 1)$. If an ad is shown 10 times and clicked 3 times, the posterior is

$$\text{Beta}(1 + 3, 1 + 7) = \text{Beta}(4, 8).$$

6.7 Thompson sampling

Referenced concepts. Used here: reward R / r_t (Section 1.3); UCB (Section 6.5); Bayesian updating (Section 6.6).

Thompson sampling samples a possible value of each arm from its posterior and plays the arm with the highest sampled value.

```
for each time step:
    for each arm a:
        sample theta_a from posterior of arm a
    play arm with largest sampled theta_a
    observe reward
    update posterior of played arm
```

Takeaway. UCB uses optimistic confidence bounds. Thompson sampling uses random plausible worlds from the posterior. Both explore more when uncertainty is high.

Chapter 7

Computational Learning Theory

7.1 Main goals of COLT

Computational Learning Theory (COLT) studies when learning is possible and how expensive it is. It focuses on whole families of learning problems, not on one trained model.

Its central questions are:

- *sample complexity*: how many examples are needed;
- *computational efficiency*: how much computation is needed;
- *guarantees*: what can be promised for all target concepts from a chosen class.

> **Takeaway.** COLT is a language for saying things like: "this whole class of rules can be learned with polynomially many examples", or "this class is too expressive for this kind of guarantee". In this course, it is mostly applied to simple symbolic classes such as conjunctions, CNF/DNF formulas, decision trees, finite hypothesis classes, and VC dimension examples.

7.1.1 Map of the COLT chapter

The chapter has one intended chain:

$$\text{concept learning} \to \text{learning model} \to \text{mistake-bound algorithms} \to \text{reductions} \to \text{PAC and VC guarantees}$$

Part	Role in the story
Concept learning	Defines the basic binary classification task: examples are positive or negative.
Concept and hypothesis classes	Separate what the true rule may be from what the learner can output.
Learning models	Define what successful learning means. This course uses mistake-bound learning and PAC learning.
Mistake-bound algorithms	Concrete online learners with provable bounds: conjunction learner, halving, Winnow.
Reductions	Reuse a known learner, usually the conjunction learner, to learn richer formula classes.

PAC learning	Gives statistical generalization guarantees from random i.i.d. samples.
VC dimension	Controls PAC sample complexity when the hypothesis class is infinite.

ELI5. Think of COLT as a sequence of questions. First: what rule are we trying to learn? Second: what counts as success? Third: do we have an algorithm with a proof? Fourth: can we reuse the proof for other rule classes?

7.2 Concept learning

Referenced concepts. Used here: learning models (Section 7.3); mistake-bound learning (Section 7.4); PAC learning (Section 7.7).

Concept learning is the basic setting used in this COLT block. It is a simplified binary classification problem. The learner sees examples and must learn a rule that decides whether a new example is positive or negative.

7.2.1 Instance space, concept, and concept class

Definition 7.1 (Instance space)**.**
The instance space X is the set of all possible examples.

Definition 7.2 (Concept)**.**
A concept $C \subseteq X$ is the set of positive examples. Equivalently, it is a binary function

$$c : X \to \{0, 1\},$$

where $c(x) = 1$ means that x is positive and $c(x) = 0$ means that x is negative.

Definition 7.3 (Concept class)**.**
A concept class $\mathcal{C}$ is a set of possible target concepts. It says what kind of hidden rule the learner is allowed to assume.

Example 7.1 (A conjunction concept class)**.**
Let $X = \{0, 1\}^n$. One possible target concept is the set of all tuples satisfying

$$x_1 \wedge \neg x_3.$$

This means: positive examples are exactly those binary vectors where the first feature is 1 and the third feature is 0. If the concept class is "all conjunctions", then formulas of this kind are allowed target concepts.

7.2.2 Hypothesis and hypothesis class

Definition 7.4 (Hypothesis and hypothesis class)**.**
A *hypothesis* h is the learner's proposed decision rule: a finite description, such as a formula, decision tree, list of rules, linear classifier, or neural network with weights, that implements a function

$$h : X \to \{0, 1\}.$$

The hypothesis class $\mathcal{H}$ is the set of all hypotheses the learner is allowed to output.

Symbol guide. X = instance space, the set of possible examples.
$x \in X$ = one example.
$C \subseteq X$ = true concept, meaning the true set of positive examples.
$c : X \to \{0, 1\}$ = indicator function of the true concept.
$h : X \to \{0, 1\}$ = hypothesis/model output by the learner. It is the learner's guess of c.
$\mathcal{C}$ = concept class, the family of possible true target concepts.
$\mathcal{H}$ = hypothesis class, the family of possible guesses the learner can express.

Takeaway. In this chapter, the word *model* usually means the same practical thing as *hypothesis*: a concrete decision rule the learner can output. It is not a new mathematical object. It is the learned rule that tries to imitate the unknown true concept.

Example 7.2 (Concept versus hypothesis)**.**
Suppose X is a set of animals described by binary features, and the hidden target concept is

$$C = \{\text{all mammals}\}.$$

The learner does not directly see the set C. It only receives labeled examples. After learning, it might output the hypothesis

$$h(x) = \text{“has hair”}.$$

This hypothesis is not literally the same object as the concept C. It is a finite rule that tries to classify the same examples as C. If it classifies every animal correctly, then it represents the target concept perfectly. If it makes mistakes, it is only an approximation.

7.2.3 Expressiveness: many concepts, fewer hypotheses

A concept is any possible subset of X. A hypothesis must be something the learner can actually write down in its chosen output language. This can be much more restrictive.

For Boolean inputs $X = \{0, 1\}^n$, there are 2^n possible inputs. A completely arbitrary concept independently labels each input as positive or negative, so the number of possible concepts is

$$2^{2^n}.$$

If the learner is only allowed to output conjunctions such as

$$x_1 \wedge \neg x_3 \wedge x_7,$$

then each variable has only a few roles in the formula: use it positively, use it negated, or do not use it. This gives roughly 3^n conjunction-like hypotheses, not 2^{2^n} arbitrary concepts.

Takeaway. The hypothesis class is much smaller than the set of all possible concepts. The limitation is structural: many possible target concepts cannot be represented by conjunctions at all, no matter how many training examples are provided.

7.3 Learning models

Referenced concepts. Uses: concept learning (Section 7.2). Used later: mistake-bound learning (Section 7.4); PAC learning (Section 7.7).

A learning model defines the rules of the learning game:

- how examples arrive;
- what the learner must output;
- how success is measured;
- what assumptions are made about the data.

This course uses two main learning models:

1. **mistake-bound learning**: online prediction, possibly adversarial order, count mistakes;
2. **PAC learning**: random i.i.d. training sample, bound future error with high probability.

Example 7.3 (Same concept class, different learning model)**.**
Consider conjunctions over Boolean variables. In the mistake-bound model, we ask how many online mistakes a conjunction learner can make before it stabilizes. In the PAC model, we ask how many random labeled examples are enough so that the learned conjunction has low error on future examples.

Takeaway. Concept learning says what kind of task we study. A learning model says what kind of guarantee we want for that task.

7.4 Mistake-bound learning

Referenced concepts. Uses: concept class and hypothesis class (Section 7.2); learning models (Section 7.3). Used later: concrete mistake-bound algorithms (Section 7.5); MB ⇒ PAC (Section 7.7.5.3).

7.4.1 Online setting

In the mistake-bound model, learning is online:

1. an example $x \in X$ arrives;
2. the learner predicts its label;
3. the true label is revealed;
4. the learner updates if needed.

The examples need not be i.i.d. and need not arrive in a friendly order. They may be chosen adversarially.

7.4.2 Mistake bound

An algorithm learns a concept class $\mathcal{C}$ in the mistake-bound model if, for every target concept $C \in \mathcal{C}$, the total number of mistakes is bounded by a polynomial in the instance size n.

Definition 7.5 (Mistake bound)**.**
A mistake bound is an upper bound on the number of wrong predictions the algorithm can make, no matter which target concept from $\mathcal{C}$ is used and no matter in which order examples are presented.

> **ELI5.** A teacher shows flashcards one by one. The learner must answer each time. We count wrong answers. A small mistake bound says: even if the teacher chooses difficult flashcards, the learner cannot be wrong too many times.

7.4.3 Efficient mistake-bound learning

Mistake-bound learnability only bounds the number of mistakes. Efficient mistake-bound learning also requires the computation per example to be polynomial in n.

> **Takeaway.** Mistake-bound learning has two separate questions: how many mistakes can happen, and how expensive each update/prediction is. A method can have a small mistake bound but still be computationally impractical if it stores too many hypotheses.

7.5 Concrete mistake-bound algorithms

> **Referenced concepts.** Uses: mistake-bound model (Section 7.4). Used later: reductions between concept classes (Section 7.6); PAC via mistake-bound learners (Section 7.7.5.3).

This section gives concrete algorithms that fit the mistake-bound model. They are examples of the same pattern: maintain some internal hypothesis, make predictions, and update the hypothesis after mistakes.

7.5.1 Generalization algorithm for conjunctions

For Boolean variables $x_1, \dots, x_n$, start with the most specific conjunction containing all literals:

$$x_1 \wedge \neg x_1 \wedge x_2 \wedge \neg x_2 \wedge \cdots \wedge x_n \wedge \neg x_n.$$

This initial hypothesis is contradictory, so it predicts no positive examples. When a positive example arrives, remove every literal that is false for that example. The remaining literals are the conditions that all observed positive examples still satisfy.

```
initialize h = x1 and not x1 and ... and xn and not xn
for each labeled example (x, y):
    predict h(x)
    if y = 1 and h(x) = 0:
        remove from h all literals that are false on x
    if y = 0 and h(x) = 1:
        report that the target is not representable by a conjunction
```

The mistake bound is at most $n + 1$ in the usual analysis. Intuitively, after the first positive mistake the hypothesis loses many impossible literals, and after that each positive mistake removes at least one remaining wrong literal.

Example 7.4 (Conjunction learner on four variables)**.**
Start with

$$h_0 = p_1 \wedge \neg p_1 \wedge p_2 \wedge \neg p_2 \wedge p_3 \wedge \neg p_3 \wedge p_4 \wedge \neg p_4.$$

Suppose the first positive example is

$$x_1 = (1, 0, 0, 1), \qquad y_1 = 1.$$

A positive example must satisfy the target conjunction, so we delete all literals false on this example. The hypothesis becomes

$$h_1 = p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge p_4.$$

If later another positive example is

$$x_3 = (0, 1, 1, 1), \qquad y_3 = 1,$$

then the only literal from h_1 that is true on both positive examples is p_4, so

$$h_3 = p_4.$$

> **Exam use.** For conjunction-learning tasks, write the current hypothesis after each example. Positive examples remove impossible literals. Negative examples usually do not remove literals in this simple generalization algorithm, unless the current hypothesis wrongly predicts positive.

7.5.2 Halving algorithm

The halving algorithm works with a finite hypothesis class $\mathcal{H}$. It keeps the *version space*: all hypotheses still consistent with all observed labels.

```
initialize version_space = H
for each example x:
    predict by majority vote of hypotheses in version_space
    receive true label y
    delete every hypothesis that predicts x incorrectly
```

If the algorithm makes a mistake, then at least half of the remaining hypotheses voted incorrectly. Therefore at least half are deleted. This gives the mistake bound

$$M \leq \log_2 |\mathcal{H}|.$$

Example 7.5 (Halving with conjunctions on three variables)**.**
For all non-contradictory conjunctions on three variables, each variable can be absent, positive, or negated. Thus

$$|\mathcal{H}| = 3^3 = 27.$$

If the first majority prediction is wrong, at least half of the hypotheses are inconsistent with the new label and are deleted. At most

$$\left\lfloor \frac{27}{2} \right\rfloor = 13$$

hypotheses remain.

> **Takeaway.** Halving is conceptually simple and gives a clean mistake bound. It is often not efficient, because storing and voting over the whole hypothesis class can be enormous.

7.5.3 Winnow

Winnow is another mistake-bound algorithm. In this course, the important fact is its target class and bound, not the full update derivation.

Winnow is designed for monotone k-disjunctions, such as

$$x_2 \vee x_7 \vee x_{10},$$

where only a small number k of variables are relevant. Its mistake bound is

$$M \leq 2 + 2k \log_2 n.$$

This can be much better than bounds that scale linearly with n when $k \ll n$.

> **Takeaway.** Conjunction learning is the basic symbolic learner. Halving is the clean theoretical learner. Winnow is important because it shows that sparse rules can be learned with a bound depending only logarithmically on the number of variables.

7.6 Reductions between concept classes

> **Referenced concepts.** Uses: conjunction learner (Section 7.5.1); mistake-bound learning (Section 7.4); hypothesis class (Section 7.2.2). Used later: proper/improper learning (Section 7.6.3); exam traps (Section 7.9).

Reductions explain why the concrete algorithms above matter beyond their first target class. If we can transform examples for a new class into examples for an already learnable class, then we can reuse the existing learner.

7.6.1 Learning disjunctions via conjunctions

A disjunction can be learned by learning the complement with a conjunction learner and then negating the result. The high-level idea is De Morgan duality:

$$\neg(a \vee b) = \neg a \wedge \neg b.$$

So a learner for conjunctions can be reused for disjunction-like classes after complementing the labels and translating the final hypothesis back.

7.6.2 k-CNF and k-DNF idea

This part is a standard COLT trick: reduce learning one logical formula class to learning a simpler formula class over newly created features.

> **Symbol guide.** A *literal* is a variable or its negation, for example x_1 or $\neg x_3$.
> A *clause* is an OR of literals, for example $(x_1 \vee \neg x_3 \vee x_4)$.
> A *term* is an AND of literals, for example $(x_1 \wedge \neg x_3 \wedge x_4)$.
> A *CNF formula* is an AND of clauses. Example: $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_4)$.
> A *DNF formula* is an OR of terms. Example: $(x_1 \wedge \neg x_3) \vee (x_2 \wedge x_4)$.
> k-CNF means that each clause has at most k literals. k-DNF means that each term has at most k literals.

For fixed k, there are only polynomially many possible clauses of size at most k over n Boolean variables. We can therefore list all possible small clauses and create one new artificial variable for each clause. Such a variable is a *meta-variable*: a computed feature saying whether the whole clause is true.

For the clause $(x_1 \vee \neg x_3)$, the meta-variable $z_{1,\neg 3}$ is defined by

$$z_{1,\neg 3}(x) = 1 \quad \text{iff} \quad (x_1 \vee \neg x_3) \text{ is true on } x.$$

Now consider a k-CNF target such as

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_4).$$

After introducing meta-variables for clauses, it becomes

$$z_{1,\neg 3} \wedge z_{\neg 2,4}.$$

This is a monotone conjunction: an AND of variables without negations, for example

$$z_1 \wedge z_4 \wedge z_9.$$

> **Takeaway.** A k-CNF formula is an AND of small OR-clauses. If every possible small OR-clause is turned into one meta-variable, then learning a k-CNF becomes learning which meta-variables must appear in one monotone conjunction. This is why the earlier conjunction learner is useful here.

The same idea can be used for k-DNF by duality: a k-DNF is an OR of small AND-terms. The course mainly wants the structural idea, not a long derivation.

7.6.3 Proper and improper learning

Proper learning means the learner outputs a hypothesis from the same class as the target concept. Improper learning allows the learner to output from a different class, as long as it represents the target correctly.

Example 7.6 (Improper learning)**.**
Learning k-term DNF using k-CNF hypotheses is improper learning: the target class is DNF, but the output language is CNF.

> **Takeaway.** Reductions are useful because they let us prove learnability of new concept classes without inventing a new learner from zero. But the output class matters: if the output formula is in a different syntactic class, the result is improper learning.

7.7 PAC learning

> **Referenced concepts.** Uses: concept learning (Section 7.2); hypothesis class (Section 7.2.2); mistake-bound learning (Section 7.4); VC dimension (Section 7.8).

7.7.1 PAC as a different learning model

Mistake-bound learning counts online mistakes. PAC learning asks a different question: after seeing a random training sample, will the final hypothesis generalize to new examples?

PAC means *Probably Approximately Correct*. The learner receives a batch of examples and should output a hypothesis whose true error is at most ε with probability at least $1 - \delta$.

7.7.2 Assumptions required for PAC

The basic PAC model assumes:

1. There is an instance space X.
2. There is a hidden target concept $C \subseteq X$, equivalently a true labeling function $c : X \to \{0, 1\}$.
3. The target concept belongs to a known concept class $\mathcal{C}$.
4. Training examples are drawn i.i.d. from a fixed but unknown distribution D over X.
5. Train and test examples come from the same distribution D.
6. Labels are correct/noise-free in the basic version: each sampled x receives label $c(x)$.
7. The learner outputs a hypothesis h from some hypothesis class $\mathcal{H}$.
8. Error is measured under the same distribution:

$$\mathrm{err}(h) = \mathbb{P}_{x \sim D}[h(x) \neq c(x)].$$

> **Common trap.** PAC does not say that the training set contains all possible inputs. It says that, under the same distribution that generated the training examples, the learned hypothesis will probably have small error on fresh examples.

7.7.3 Meaning of ε and δ

The parameters have different roles:

- ε is the allowed true error.
- δ is the allowed probability that learning fails.

Example 7.7 (Reading ε and δ)**.**
If

$$\varepsilon = 0.05, \qquad \delta = 0.01,$$

then the guarantee says: with probability at least 99%, the learned hypothesis is wrong on at most 5% of future examples drawn from the same distribution.

> **ELI5.** "Probably" is controlled by δ. "Approximately correct" is controlled by ε.

7.7.4 What PAC is useful for

PAC learning is not an algorithm like Q-learning or the conjunction learner. It is a framework for proving guarantees. It is useful for answering:

- how many examples are sufficient;

- whether a concept class is learnable at all;
- whether it is learnable efficiently;
- whether a known learner in one model, such as mistake-bound learning, also implies a PAC guarantee.

> **Takeaway.** A dataset is not "PAC". A concept class and a learning algorithm can have a PAC guarantee. The guarantee says that enough random examples are sufficient for low future error with high probability.

7.7.5 How PAC-learnability is usually proved

For this course, there are three common proof patterns.

7.7.5.1 Finite hypothesis class bound

If $\mathcal{H}$ is finite and the learner returns a hypothesis consistent with the training data, a typical sufficient sample bound is

$$m \geq \frac{1}{\varepsilon} \ln \frac{|\mathcal{H}|}{\delta}.$$

This says: more possible hypotheses require more examples; stricter accuracy and confidence also require more examples.

Example 7.8 (Small finite class)**.**
Suppose $|\mathcal{H}| = 1000$, $\varepsilon = 0.1$, and $\delta = 0.05$. Then

$$m \geq \frac{1}{0.1} \ln \frac{1000}{0.05} = 10 \ln(20000) \approx 99.0.$$

So about 100 examples are enough according to this bound. This does not mean 100 examples are always practically ideal; it means the theoretical sufficient bound is polynomial and finite.

7.7.5.2 Finite VC dimension

If $\mathcal{H}$ is infinite, $|\mathcal{H}|$ is not useful. Then VC dimension measures the effective capacity of the class. Small VC dimension gives PAC sample bounds even when $\mathcal{H}$ contains infinitely many hypotheses. VC dimension is introduced in Section 7.8.

7.7.5.3 Mistake-bound learner implies PAC learner

If a class has a mistake-bound learner with polynomial mistake bound M, then it can be converted into a PAC learner. The important intuition is: if the learner cannot make many mistakes in the adversarial online setting, then it can also learn from enough random examples.

A typical bound has the form

$$m = O\left(\frac{M}{\varepsilon} \ln \frac{M}{\delta}\right).$$

Example 7.9 (Conjunctions)**.**
The conjunction learner has mistake bound at most $n + 1$. Therefore conjunctions are not just learnable in the mistake-bound model; they also have a PAC-style learnability guarantee. In PAC language, enough i.i.d. examples are sufficient to probably output a conjunction with low future error.

7.8 Shattering and VC dimension

Referenced concepts. Uses: PAC learning (Section 7.7); hypothesis class (Section 7.2.2). Used for: PAC sample bounds when raw $|\mathcal{H}|$ is infinite.

7.8.1 Shattering

Definition 7.6 (Shattering)**.**
A hypothesis class shatters a finite set of examples if it can realize every possible labeling of that set.

For example, shattering three points means that for all $2^3 = 8$ possible positive/negative labelings of those points, some hypothesis from the class matches that labeling.

7.8.2 VC dimension

Definition 7.7 (VC dimension)**.**
$\mathrm{VC}(\mathcal{H})$ is the size of the largest set shattered by $\mathcal{H}$.

ELI5. If a model class can color any pattern of red/blue labels on a set of points, it has enough flexibility to shatter those points. VC dimension measures this flexibility.

Example 7.10 (Lines in the plane)**.**
Half-planes in $\mathbb{R}^2$ can shatter some sets of three points, but cannot shatter every set of four points. Their VC dimension is 3.

7.8.3 Why VC dimension matters for PAC

Finite $|\mathcal{H}|$ bounds are useless for infinite hypothesis classes. VC dimension replaces raw counting with a capacity measure. If $\mathrm{VC}(\mathcal{H})$ is small enough, PAC learning can still be possible even when $\mathcal{H}$ contains infinitely many hypotheses.

Takeaway. Finite $\mathcal{H}$: count hypotheses. Infinite $\mathcal{H}$: use VC dimension. Both are ways to control how many examples are needed for generalization.

7.9 Relations and exam traps

7.9.1 Mistake-bound versus PAC

	Mistake-bound learning	PAC learning
Examples	online sequence	i.i.d. training sample
Order	may be adversarial	random from fixed distribution
Success measure	number of online mistakes	future error with high probability
Typical output	updated during prediction	final hypothesis after training sample

7.9.2 Subset relations between concept classes

Here $\mathcal{C}_{small}$ and $\mathcal{C}_{big}$ are concept classes, meaning sets of possible target concepts. The notation

$$\mathcal{C}_{small} \subseteq \mathcal{C}_{big}$$

means that every target concept from the small class is also contained in the big class. For instance, conjunctions are a subset of all Boolean formulas.

The safe directions are:

- If the big class is learnable, then the small class is also learnable by using the same learner.
- If the small class is learnable, this does not automatically imply that the big class is learnable.

This is different from reductions in Section 7.6. A subset relation compares class sizes. A reduction transforms one learning problem into another.

7.9.3 Proper-learning caution

Proper learning was defined in Section 7.6.3. The exam trap is that learnability statements depend on both the target class $\mathcal{C}$ and the allowed output class $\mathcal{H}$.

> **Takeaway.** Always identify three things: the target concept class $\mathcal{C}$, the allowed hypothesis/output class $\mathcal{H}$, and the learning model. Many COLT exam traps hide in mixing these three.

Part II

Natural Language Processing

The NLP part starts a new application area, but the learning logic remains similar: define the object being estimated, choose a representation, and learn from data. The storyline is:

$$\text{probabilities over text} \to \text{sparse count vectors} \to \text{matrix structure} \to \text{neural dense/contextual representations}$$

Chapter 8

NLP 1: Probabilistic Models

8.1 NLP chapter map

The NLP part is easiest to read as a progression of representations. The course starts with words as symbols and counts, then turns text into vectors and matrices, and finally uses neural models that learn representations by prediction.

Block	Role in the story
Probabilistic models	Model text using probabilities: sentence probability, next-word probability, and Naive Bayes classification.
Vector models	Represent documents and words as vectors so similarity can be measured geometrically.
Matrix/topic models	Stack many vectors into a matrix and explain a corpus by hidden lower-dimensional structure. These are classic interpretable corpus-analysis methods, not the core mechanism of LLMs.
Neural models	Learn embeddings and contextual representations directly from prediction/classification objectives.

Takeaway. The same text can be viewed in several ways: as a probability distribution, as a bag of words, as a sparse vector, as a row in a matrix, as a mixture of topics, or as a sequence of contextual neural vectors.

8.2 What NLP is about

Referenced concepts. Used here: language model (Section 8.3); topic model (Section 10.2); embedding (Section 9.6); Transformer (Section 11.7).

Natural Language Processing studies how computers process human language. Typical tasks include language modeling, text classification, topic modeling, information extraction, machine translation, question answering, and conversational agents.

Exam use. For NLP exam answers, first identify the representation: probabilities, word counts, sparse vectors, dense embeddings, topic mixtures, or contextual Transformer vectors. Most confusion comes from mixing these representations.

8.3 Language modeling

A language model assigns probability to a word/token sequence

$$P(w_1, \ldots, w_n)$$

or predicts the next word/token

$$P(w_n \mid w_1, \ldots, w_{n-1}).$$

Symbol guide. w_i = the i-th word or token.
$W = (w_1, \ldots, w_n)$ = a full word/token sequence.
$P(W)$ = probability of the whole sequence.
$P(w_i \mid \text{context})$ = probability of the next token under a language model.

ELI5. A language model is autocomplete with probabilities. Given previous words, it assigns probabilities to possible next words. A good model gives high probability to natural continuations and low probability to strange continuations.

8.4 Chain rule and Markov assumption

Referenced concepts. Uses: language model (Section 8.3). Used later: N-gram models (Section 8.5); CNN/RNN/Transformer comparison (Sections 11.4, 11.5, 11.7).

The exact chain rule is

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1, \ldots, w_{i-1}).$$

This is usually impossible to estimate directly for long histories, because there are too many possible histories. The Markov assumption approximates the full history by only the last k words:

$$P(w_i \mid w_1, \ldots, w_{i-1}) \approx P(w_i \mid w_{i-k}, \ldots, w_{i-1}).$$

Takeaway. The Markov assumption is the bridge from the exact but impossible language model to practical N-gram models.

8.5 N-gram models

Referenced concepts. Uses: Markov assumption (Section 8.4). Used later: smoothing/log-space (Section 8.8); neural language models (Section 11.1); CNN receptive fields (Section 11.4).

An N-gram model estimates next-word probabilities by counting short contexts.

- Unigram: no context, $P(w_i)$.
- Bigram: previous word, $P(w_i \mid w_{i-1})$.
- Trigram: previous two words, $P(w_i \mid w_{i-2}, w_{i-1})$.

Maximum-likelihood bigram estimate:

$$P(w_i \mid w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}.$$

Common trap. Unseen N-grams get zero probability without smoothing. One zero can make the entire sentence probability zero.

Takeaway. N-grams are easy to estimate, but they treat words mostly as symbols and suffer from sparsity. Vector models keep the data-driven idea but represent text geometrically.

8.6 Perplexity

Referenced concepts. Uses: language model (Section 8.3); N-gram models (Section 8.5).

Perplexity measures how surprised a language model is by test text:

$$\text{PP}(W) = \left(\prod_{i=1}^{n} P(w_i \mid context_i) \right)^{-1/n}.$$

Lower perplexity is better.

ELI5. If the model consistently gives high probability to the correct next word, perplexity is low. Roughly, perplexity behaves like the average number of plausible next-word choices the model feels it has.

8.7 Naive Bayes for text

Referenced concepts. Uses: language model (Section 8.3); smoothing/log-space (Section 8.8); feature representation (Section 5.2). Related later: bag-of-words vectors (Section 9.2).

Naive Bayes uses word probabilities for classification. For a document represented as words or counts, it predicts

$$\hat{c} = \text{argmax}_c \, P(c) \prod_i P(w_i \mid c).$$

It uses a bag-of-words representation and assumes words are conditionally independent given the class.

Symbol guide. c = class label, e.g. positive or negative.
$P(c)$ = prior probability of class c.
w_i = word/token feature from the document.
$P(w_i \mid c)$ = probability of seeing word w_i in class c.
$\hat{c}$ = predicted class.

Definition 8.1 (Bag of words)**.**
A bag-of-words representation keeps word counts or word occurrences and ignores order.

ELI5. Put all words from the text into a bag and shake it. Naive Bayes does not care about the original order; it cares which words are in the bag and how often.

8.8 Smoothing and log-space

Referenced concepts. Uses: N-gram models (Section 8.5); Naive Bayes (Section 8.7).

Laplace smoothing prevents zero probabilities:

$$P(w \mid c) = \frac{\text{count}(w, c) + 1}{\sum_{w'} \text{count}(w', c) + |V|}.$$

For long documents use logs:

$$\log P(c) + \sum_i \log P(w_i \mid c).$$

Exam use. Naive Bayes exam tasks often ask: write the formula, count priors and likelihoods, classify a small test document, and explain underflow/log probabilities.

Chapter 9

NLP 2: Vector Models and Embeddings

Probabilistic NLP counted words and word histories. Vector models keep the data-driven idea but change the representation: documents and words become vectors that can be compared by geometry.

9.1 Distributional semantics

> **Referenced concepts.** Used later: term-context matrix (Section 9.2.2); word2vec/skip-gram (Section 9.5); embeddings (Section 9.6).

Distributional semantics is based on the idea that words with similar contexts tend to have similar meanings.

> **ELI5.** You can guess what an unknown word means by looking at the words around it. If it appears near "garlic", "rice", and "leafy", it may be a vegetable.

9.2 Sparse count vectors

Sparse vector models are the first geometric representation of text. They still use counts, but the counts become coordinates in a vector space.

9.2.1 Term-document matrix

A document can be represented as a vector of word counts. Stacking such vectors gives a term-document matrix: rows may represent documents and columns may represent terms (or the transpose, depending on convention). This representation is fundamental in information retrieval.

9.2.2 Term-context matrix

A word can also be represented by the words that occur near it. This gives a word-word or term-context matrix. Similar words have similar context vectors.

9.2.3 Cosine similarity

Cosine similarity compares the direction of two vectors:

$$\cos(x, y) = \frac{x \cdot y}{\|x\|\|y\|}.$$

It is often used because document length should not dominate similarity.

> **Takeaway.** Count vectors make text measurable. A document becomes a point in a high-dimensional word space, and a word can become a point in a context space.

9.3 TF-IDF

> **Referenced concepts.** Uses: term-document matrix (Section 9.2.1); bag of words (Section 8.7). Used later: matrix factorization/topic models (Sections 10.3, 10.4).

TF-IDF weights words by being frequent in a document but not too common in all documents:

$$tfidf(t, d) = tf(t, d) \cdot idf(t),$$

where

$$idf(t) = \log \frac{N}{df(t)}.$$

Here N is the number of documents and $df(t)$ is the number of documents containing term t.

> **Takeaway.** TF-IDF is still a sparse symbolic representation. It improves raw counts by reducing the importance of very common words.

9.4 Sparse versus dense vectors

> **Referenced concepts.** Uses: TF-IDF (Section 9.3). Used later: embeddings (Section 9.6); neural embedding matrix (Section 11.2).

Sparse vectors have one dimension per word/context/document feature. Most entries are zero. Dense embeddings have far fewer dimensions and try to encode useful semantic information.

> **Takeaway.** Sparse vectors are interpretable but high-dimensional. Dense vectors are harder to inspect but usually generalize better because similar words can have nearby vectors.

9.5 word2vec and skip-gram

Referenced concepts. Uses: distributional semantics (Section 9.1); dense embeddings (Section 9.4); neural prediction objective (Section 11.3).

Skip-gram trains embeddings by predicting context words from a center word. It turns distributional semantics into a prediction task.

ELI5. Skip-gram gives the model a small game: from a center word, guess nearby words. To play well, it must put words with similar contexts into similar vectors. The useful result is not the game score, but the vectors learned while playing.

9.6 Static versus contextual embeddings

Referenced concepts. Uses: sparse/dense vector distinction (Section 9.4). Used later: Transformer representations (Section 11.7); query/key/value attention (Section 11.7.4).

Old embeddings such as word2vec assign one vector to each word type. Modern Transformer models start with token embeddings but then transform them into contextual representations.

Example 9.1 (Static embedding versus contextual representation)**.**
Consider the phrases

- "river bank";
- "bank loan".

A static embedding model gives the word "bank" one fixed vector. A Transformer may start from the same token embedding for "bank", but after self-attention the contextual representation becomes different: in the first phrase it is influenced by "river", while in the second phrase it is influenced by "loan".

Takeaway. Vector models lead naturally to neural embeddings: instead of manually counting contexts, a model can learn useful dense vectors from a prediction objective.

Chapter 10

NLP 3: Matrix Models and Topic Models

Vector models gave us sparse or dense vectors for words and documents. This chapter uses the sparse document-word view to ask a corpus-level question: can a large collection of documents be explained by a smaller number of hidden themes?

> **Takeaway.** This chapter is about classic corpus analysis. LSA, NMF, and LDA are older than modern LLMs and are not the core mechanism of Transformers. They are still useful because they give explicit, inspectable structure: topics, components, or latent dimensions that help us explore a document collection.

10.1 The common pipeline

> **Referenced concepts.** Uses: term-document matrix (Section 9.2.1); TF-IDF (Section 9.3); sparse vectors (Section 9.2). Used later: LSA/SVD (Section 10.3); NMF (Section 10.4); LDA (Section 10.5).

The methods in this chapter usually start with the same basic pipeline:

raw documents $\longrightarrow$ document-term matrix X $\longrightarrow$ hidden lower-dimensional structure.

This is the unifying story of the chapter. LSA/SVD, NMF, and LDA are not random unrelated methods. They are three classic ways of asking a similar question:

Can many documents be explained by fewer hidden themes/components?

The word "topic" is used most cleanly for LDA and NMF. For LSA it is safer to say *topic-like latent semantic dimension*, because SVD dimensions are mathematical directions and may not correspond to one clean human-readable topic.

The matrix X is built from text. A common convention is:

- rows = documents;
- columns = words/terms;
- entries = word counts, binary presence values, or TF-IDF weights.

For example:

	goal	team	data	model	doctor
d_1	2	1	0	0	0
d_2	1	2	0	0	0
d_3	0	0	3	1	0
d_4	0	0	0	1	2

This matrix is usually large and sparse. Matrix/topic methods try to replace it by a smaller explanation. Instead of saying "document d_1 contains word 1, word 2, word 350, ...", we want to say something like "document d_1 is mostly about sport", or more cautiously, "document d_1 has a high weight on hidden component 1".

> **Common trap.** LSA, NMF, and LDA do not read raw text directly. Text must first be converted into a numeric representation. For LSA and NMF this is explicitly a matrix, usually a document-term matrix. LDA is usually described as a probabilistic model of word counts, but it also works from the observed words/counts in a document collection.

10.2 Topic modeling

> **Referenced concepts.** Uses: document-term matrix (Section 10.1); unsupervised learning idea from COLT/PAC contrast (Section 7.3). Used later: NMF (Section 10.4); LDA (Section 10.5); comparison with LLMs (Section 10.6).

Topic modeling is unsupervised learning for document collections. It tries to discover hidden themes from word co-occurrence patterns.

The important point is that the topic names are usually *not predefined*. The user usually chooses only the number of topics/components K. The algorithm then learns K word patterns. Humans inspect the important words and give the topics names afterwards.

Input:

- a collection of documents;
- a vocabulary;
- a chosen number of topics/components K.

Typical output:

- topics/components represented by important words;
- for each document, a vector saying how much each topic appears in that document.

Example 10.1 (Learned topics are named after training)**.**
A topic model trained on news articles might learn these word groups:

- Topic 1: election, party, vote, parliament, candidate;
- Topic 2: bank, inflation, market, price, interest;
- Topic 3: team, match, goal, league, player.

The algorithm usually does not output the labels "politics", "economy", and "sport". A human assigns those names after seeing the top words.

This is different from supervised text classification. In supervised classification, labels such as "sport" or "technology" are given in advance and the model learns to predict them. In topic modeling, the model discovers hidden themes without being given their names.

ELI5. A document is like soup made from several flavors. Topic modeling tries to discover the flavors and estimate how much of each flavor each document contains. The names of the flavors are added by the cook afterwards.

10.3 LSA and SVD

Referenced concepts. Uses: document-term matrix (Section 10.1); TF-IDF (Section 9.3); cosine similarity (Section 9.2.3). Compare with: NMF (Section 10.4); LDA (Section 10.5).

Latent Semantic Analysis is the simplest matrix method in this chapter. It is one way to find topic-like hidden semantic structure in a text collection, but it does this as matrix compression, not as a probabilistic topic model.

LSA takes a text matrix X, usually a document-term or term-document matrix with count or TF-IDF entries, and applies truncated Singular Value Decomposition:

$$X \approx U_k \Sigma_k V_k^T.$$

The purpose is to compress the original sparse word space into a smaller latent space. Instead of representing a document by thousands of word-count dimensions, LSA may represent it by k latent semantic dimensions.

Symbol guide. X = original document-term or term-document matrix.
k = chosen number of latent dimensions.
U_k, Σ_k, V_k^T = truncated SVD factors.
The latent dimensions are learned from the matrix. They are not manually named and are not guaranteed to be clean topics.

Takeaway. LSA/SVD can be used to find *topic-like* semantic dimensions in text, but it is not a pure topic model. In the course slides it is discussed as topic modelling via matrix factorization, but the safer wording is: LSA finds latent semantic dimensions that may behave like topics. These dimensions may mix several meanings and can have negative coordinates, so they are often harder to name than LDA or NMF topics.

LSA is useful for dimensionality reduction, information retrieval, document similarity, and rough semantic structure discovery.

Common trap. SVD is a general matrix decomposition and can be applied to many numeric matrices, including dense embedding matrices. In this course, however, LSA means the classic NLP use of SVD on a count/TF-IDF-style text matrix.

10.4 NMF

Referenced concepts. Uses: document-term matrix (Section 10.1); topic modeling (Section 10.2); TF-IDF (Section 9.3). Compare with: LSA (Section 10.3); LDA (Section 10.5).

Non-negative Matrix Factorization is another way to find hidden topics/components in a document collection. It is often used as a topic discovery method because its components are non-negative and therefore easier to read as groups of important words.

NMF explains a document-term matrix by fewer hidden components. It assumes the matrix has no negative entries and factorizes it as

$$X \approx WH,$$

where W and H are also non-negative.

A common topic-modeling interpretation is:

- X = document-term matrix;
- W = document-topic weights;
- H = topic-word weights.

How it is used:

1. Build a non-negative document-term matrix, often with counts or TF-IDF.
2. Choose the number of components/topics K.
3. Factorize $X \approx WH$.
4. For each row/component in H, inspect the highest-weight words.
5. Name the discovered component, if it is interpretable.

> **Takeaway.** NMF is directly usable as a topic discovery method: each component can often be read as one topic, and each document is represented as an additive mixture of those components.

Example 10.2 (NMF as additive topics)**.**
If one NMF component has high weights for "team", "goal", and "match", and another has high weights for "bank", "inflation", and "market", a document can be represented as a non-negative mixture of these components. Because the values are non-negative, components add evidence instead of cancelling each other with negative numbers.

NMF is not a probabilistic model of how documents are generated. It is a matrix factorization whose non-negative factors are often easy to read as topics.

10.5 LDA idea

> **Referenced concepts.** Uses: topic modeling (Section 10.2); probability distributions (Section 8.3). Compare with: NMF (Section 10.4) and LSA/SVD (Section 10.3).

Latent Dirichlet Allocation is the most explicitly "topic model" method in this chapter. It is used to discover topics in a corpus and to represent each document as a mixture of those topics.

LDA is probabilistic: it describes a story for how documents could be generated from hidden topics.

> **Takeaway.** LDA is a method for finding topics in text. It learns two main things: which words belong to each topic, and which mixture of topics appears in each document.

In LDA:

- a topic is a probability distribution over words;
- a document is a probability distribution over topics;
- each word token in a document has a hidden topic assignment.

The generative story is:

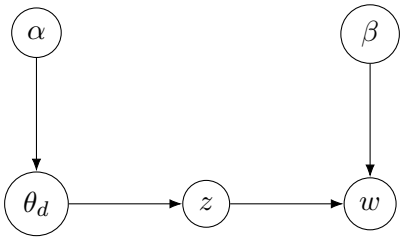
1. For each document d, choose a topic mixture θ_d.
2. For each word position in that document, choose a hidden topic z from θ_d.
3. Choose the observed word w from the word distribution of topic z.

For example, if a document has topic mixture

$$\theta_d = 0.7 \cdot \text{sport} + 0.3 \cdot \text{economy},$$

then most word positions are expected to come from the sport topic, but some may come from the economy topic. One observed word such as "goal" might be assigned to the sport topic; another observed word such as "market" might be assigned to the economy topic.

Symbol guide. K = chosen number of topics.
θ_d = topic mixture of document d, e.g. 70% topic 1 and 30% topic 4.
z = hidden topic assignment for one word position.
w = observed word.
α = prior controlling document-topic mixtures.
β = topic-word parameters/distributions, depending on notation.



10.5.1 How LDA learning works intuitively

We observe only the documents and their words. We do *not* observe:

- the document-topic mixture θ_d;
- the topic-word distributions;
- the hidden topic assignment z for each word token.

Learning means estimating all of these hidden objects from the observed words.

A useful mental model is iterative reassignment:

1. Choose the number of topics K.
2. Give word tokens initial random topic assignments.
3. Repeatedly visit word tokens and ask which topic best explains this word in this document.
4. A topic is more plausible if it is already common in this document.
5. A topic is also more plausible if this word is already common in that topic across the whole corpus.
6. After many such updates, the assignments become more stable. From them we can estimate document-topic mixtures and topic-word distributions.

For example, suppose the word "goal" appears in a document that already contains many words currently assigned to a sport-like topic. Also suppose "goal" is common in that topic across the corpus. Then LDA has two reasons to assign this occurrence of "goal" to the sport-like topic: the document context fits, and the topic-word pattern fits.

So LDA does not take predefined topics and classify words into them. It jointly estimates the topics and the document mixtures. The iterative reassignment story above is closest to Gibbs-sampling intuition. Other LDA inference methods, such as variational inference, do not literally perform the same random reassignment step, but they are solving the same hidden-variable problem. The exam-relevant idea is: documents are mixtures of topics, topics are distributions over words, and the topic of each word token is hidden and inferred.

LDA is used to explore large document collections, summarize corpora by topics, represent documents by topic proportions, and find groups of words that tend to appear together.

> **ELI5.** LDA is like sorting colored beads without seeing the colors directly. You only see words in documents. The algorithm repeatedly guesses which hidden color/topic produced each word. If a document already contains many sport words, another word in that document is more likely to be assigned to a sport topic. If the word "goal" often appears in the sport topic across the corpus, it becomes even more likely to belong there.

10.6 Are topic models used in LLMs?

> **Referenced concepts.** Uses: topic modeling (Section 10.2); contextual embeddings (Section 9.6); Transformers (Section 11.7).

Classic LSA, NMF, and LDA are not the core technology inside modern LLMs. Modern LLMs are usually Transformer neural networks trained to predict tokens and to learn contextual representations.

However, topic models are not useless. They are still useful when the goal is interpretability or corpus exploration rather than generating text. Typical uses include:

- quickly exploring what a large document collection is about;
- finding rough themes in survey answers, news articles, papers, or logs;
- representing documents by topic proportions for simple downstream models;
- analyzing datasets or model outputs;
- creating interpretable summaries of a corpus.

Modern workflows may also combine neural embeddings with topic-style analysis. For example, documents can be embedded by a neural model and then clustered; the clusters can then be described by important words. This is closer to modern embedding-based topic discovery than to classical LDA, but the high-level goal is similar: find interpretable groups of documents and words.

> **Takeaway.** For the exam, treat LSA/NMF/LDA as classic matrix/topic methods. For modern LLMs, treat Transformers and contextual embeddings as the central mechanism. The connection is conceptual: both try to find hidden structure in text, but they do it with different tools.

10.7 LSA, NMF, and LDA compared

Referenced concepts. Uses: LSA/SVD (Section 10.3); NMF (Section 10.4); LDA (Section 10.5); topic models versus LLMs (Section 10.6).

Method	Input/main object	What it gives	Topic interpretation
LSA/SVD	count or TF-IDF text matrix	dense latent semantic coordinates	topic-like semantic dimensions; not guaranteed clean topics
NMF	non-negative document-term matrix $X \approx WH$	additive components and document-component weights	often directly read as topics
LDA	documents as word counts in a probabilistic model	topic-word distributions and document-topic distributions	explicit probabilistic topics
LLMs	token sequences and learned embeddings	contextual token representations and next-token predictions	not classical topic models; topics may be implicit, not explicit

Exam use. For matrix/topic models, be ready to explain the pipeline: raw documents become a document-term matrix or word-count corpus; then LSA, NMF, or LDA finds hidden structure. Also be able to say that topics are usually learned, not predefined, and that classic topic models are not the same thing as modern LLMs.

Takeaway. The chapter is not three unrelated methods. LSA, NMF, and LDA are three classic answers to one question: how can we explain a large document-word matrix or corpus using fewer hidden dimensions/components/topics? LDA and NMF are most directly used for topic discovery; LSA is better described as latent semantic compression that can reveal topic-like structure.

Exam use. Safe exam wording: "All three methods can help discover hidden structure in a corpus. NMF and LDA are topic-discovery methods in the stronger sense. LSA/SVD finds latent semantic dimensions that may be topic-like, but they are not guaranteed to be clean topics."

Chapter 11

NLP 4: Neural Models, RNNs, and Transformers

Earlier NLP chapters introduced count-based language models, sparse vectors, matrix/topic models, and static embeddings. Neural models keep the same broad goals - predict, classify, and represent language - but learn the representations inside a trainable model.

Compared with topic models, neural models are less explicitly interpretable but much more flexible. A Transformer does not usually store a small list of human-readable topics. Instead, it repeatedly updates token vectors using context, and the resulting contextual representations are useful for prediction and generation.

11.1 Neural language models

> **Referenced concepts.** Uses: language model (Section 8.3); embeddings (Section 9.6); one-hot vectors and embedding matrix (Section 11.2). Related to: skip-gram (Section 9.5).

A neural language model predicts the next token from previous context:

$$P(w_t \mid w_1, \ldots, w_{t-1}).$$

This is the same language-modeling goal as in N-grams, but the probability is produced by a neural network instead of by a count table.

The basic pipeline is:

token IDs → embeddings → neural layers → scores over vocabulary → softmax probabilities.

> **Takeaway.** N-grams store probabilities for short symbolic histories. Neural language models learn a function that maps token histories to next-token probabilities.

11.2 One-hot vectors and embedding matrix

> **Referenced concepts.** Uses: sparse versus dense vectors (Section 9.4); static/contextual embeddings (Section 9.6). Used later: Transformers (Section 11.7).

A one-hot vector has length $|V|$ and contains one 1 and otherwise zeros. Conceptually, multiplying by an embedding matrix selects the dense vector for that token:

$$\text{one-hot}(w_i)^T E = e_i.$$

In implementation, the token ID usually just indexes a row of E.

Symbol guide. $|V|$ = vocabulary size.
$E \in \mathbb{R}^{|V| \times d}$ = embedding matrix.
d = embedding dimension.
e_i = dense embedding vector of token i.

ELI5. A token ID is like a library card number: it is only an index, not a meaning by itself. The embedding matrix is the lookup table that gives the token a useful vector.

11.3 Training objective

Referenced concepts. Uses: language model (Section 8.3); neural language model (Section 11.1); embedding matrix (Section 11.2).

Neural language models are trained to predict the correct next token. This is a multiclass classification problem over the vocabulary, usually trained by cross-entropy / negative log-likelihood:

$$L = -\log P(w_t \mid w_1, \ldots, w_{t-1}).$$

The embedding matrix is learned because it helps reduce this prediction loss.

Takeaway. Skip-gram and neural language modeling share the same broad idea: useful representations appear as a side effect of a prediction task.

11.4 CNNs for text

Referenced concepts. Uses: embeddings (Section 9.6); Markov/local context idea (Section 8.4). Compare with: RNNs (Section 11.5) and Transformers (Section 11.7).

A 1D convolution over text looks at local windows of embeddings. It is useful when local patterns are informative, for example short phrases in text classification. Stacking convolutions increases the receptive field, so later layers can use wider context. Pooling can make the representation less sensitive to exact position.

Takeaway. CNNs for text are a neural version of local-context thinking: they look for useful local patterns, but the patterns are learned rather than manually counted.

11.5 RNNs

Referenced concepts. Uses: embeddings (Section 9.6); neural language models (Section 11.1). Used later: backpropagation through time (Section 11.6); Transformer comparison (Section 11.7.7).

Recommended visual explanation: RNN video explanation.

11.5.1 Main idea

An RNN is a neural language model for sequences. It does not look at the whole sentence at once. It reads tokens one by one and keeps a hidden state.

At time t, the RNN receives:

- the current input vector x_t, usually the embedding of the current token;
- the previous hidden state h_{t-1}, which is the model's current summary of the earlier tokens.

It computes:

$$h_t = g(Uh_{t-1} + Wx_t),$$
$$y_t = f(Vh_t).$$

This means that h_t is not a stored answer from a table. It is recomputed at every time step from two things: the previous memory h_{t-1} and the current token vector x_t. The output y_t can then be used for a task, for example predicting the next token or assigning a label.

The flow is:

$$(x_1, h_0) \to h_1, \quad (x_2, h_1) \to h_2, \quad (x_3, h_2) \to h_3, \quad \dots$$

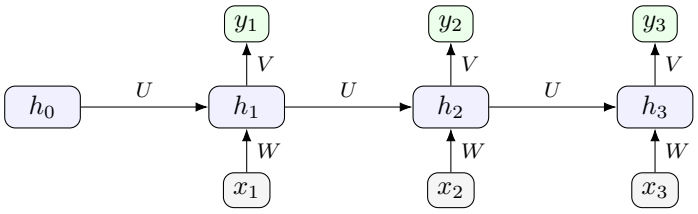
So the current token influences the current hidden state, and the current hidden state carries information to the next position.

11.5.2 What is learned in an RNN

The learned parameters are mainly the matrices U, W, and V:

- W learns how the current token embedding affects the hidden state;
- U learns how the previous hidden state affects the next hidden state;
- V learns how to turn the hidden state into output scores, for example next-token scores.

The same U, W, V are reused at every time step. This is crucial: an RNN does not learn separate parameters for position 1, position 2, position 3, and so on. It learns one repeated update rule for processing sequences of arbitrary length.



U, W, and V are learned and reused at every time step; h_1, h_2, h_3 are computed hidden states.

Symbol guide. x_t = input vector at time t, usually an embedding.
h_t = hidden state after reading token t; it is computed, not directly learned as a table.
y_t = output, for example next-token scores or class probabilities.
U = learned recurrent matrix from previous hidden state to next hidden state.
W = learned input matrix from current token vector to hidden state.
V = learned output matrix from hidden state to output.
g = hidden-state nonlinearity, such as tanh or ReLU.
f = output transformation, such as softmax for next-token probabilities.
The same U, W, V are shared across all time steps.

ELI5. An RNN is like reading with a notebook. After each word, the notebook h_t is updated. The notebook itself is different for every sentence, but the rule for updating it, represented by the learned matrices, is the same every time.

11.5.3 Why RNNs became less central

RNNs were important because they removed the fixed-window limitation of N-grams and simple CNNs. However, information must pass through the chain

$$h_1 \to h_2 \to h_3 \to \cdots \to h_t.$$

For long-distance dependencies, the useful information can become hard to preserve. Training is also sequential, because h_t depends on h_{t-1}. This is one reason why Transformers became dominant for large-scale NLP.

11.6 Backpropagation through time

Referenced concepts. Uses: RNN (Section 11.5); training objective (Section 11.3).

RNNs can be unrolled into a deep feed-forward network with shared weights. Training uses backpropagation through time. The main practical issue is that long-range dependencies are hard due to vanishing/exploding gradients.

Takeaway. RNNs solve the fixed-window limitation of N-grams and simple CNNs by carrying a state. Transformers solve a different bottleneck: they let tokens look directly at other tokens through attention.

11.7 Transformers

Referenced concepts. Uses: language model (Section 8.3); embeddings (Section 9.6); RNNs (Section 11.5). This section contains query/key/value attention (Section 11.7.4), the self-attention formula (Section 11.7.5), and positional information (Section 11.7.6).

Recommended visual explanation: Transformer video explanation.

11.7.1 Main idea

A Transformer is also a neural sequence model, but it does not pass information mainly through one recurrent hidden state. Instead, it repeatedly updates a vector for each token by letting tokens attend to other tokens.

The useful mental model is:

$$\text{one token} \quad \longrightarrow \quad \text{one vector that gets updated by context.}$$

At the start, each token has a mostly context-free embedding. After self-attention layers, the vector for that token becomes contextual: it now contains information from other relevant tokens.

A typical decoder-only language model pipeline is:

text → tokens → token embeddings+positions → Transformer blocks → next-token distribution.

11.7.2 One Transformer block as a flow

A simplified Transformer block does roughly this:

1. start with token vectors;
2. add positional information so the model knows order;
3. use self-attention so each token can pull information from other tokens;
4. use a feed-forward network to transform each token vector further;
5. repeat this block many times.

11.7.3 What is learned in a Transformer

The model learns many weight matrices. For the course-level explanation, the most important ones are:

- the token embedding matrix, which maps token IDs to vectors;
- positional embeddings or positional encoding parameters, depending on the architecture;
- attention projection matrices W_Q, W_K, W_V, which produce queries, keys, and values;
- feed-forward layer weights inside each Transformer block;
- the output layer that maps final token vectors to next-token scores.

The token vectors are not fixed after the embedding lookup. They are repeatedly updated. This is why the same token can get different contextual representations in different sentences.

> **Takeaway.** An RNN moves information through time using a hidden-state chain. A Transformer moves information inside a layer using attention between token vectors. This is why Transformers are easier to parallelize during training and why long-distance connections are more direct.

Transformers are used for modern language models, translation, summarization, question answering, classification, and many other NLP tasks.

11.7.4 Query, key, value

Referenced concepts. Uses: Transformer (Section 11.7); embeddings/contextual representations (Section 9.6). Used later: self-attention formula (Section 11.7.5).

For each current token representation x, attention computes three vectors:

$$q = xW_Q, \quad k = xW_K, \quad v = xW_V.$$

The matrices W_Q, W_K, W_V are learned. The names are easiest to understand by analogy:

- the query q means "what information is this token looking for?",
- the key k means "what kind of information does this token offer?",
- the value v means "what information should be copied/mixed if this token is selected?".

For a target token, the model compares its query to the keys of other tokens. Similar query–key pairs get high attention weight. Then the corresponding value vectors are mixed together to update the target token representation.

ELI5. For each word, attention asks: "Which other words matter for interpreting this word?" Then it blends information from those words into the current word vector.

11.7.5 Self-attention formula

Referenced concepts. Uses: query/key/value attention (Section 11.7.4); Transformer block flow (Section 11.7.2). Used next: positional information (Section 11.7.6).

The attention formula is not the final output of the model. It is the operation that *updates token vectors inside a Transformer block.*

Assume a sequence has n token vectors stacked into a matrix X. One row of X is one token representation. The model first computes

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V.$$

Then self-attention computes

$$Z = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

Here Z is the matrix of updated token vectors. Row z_i is the new contextual vector for token i.

Read the formula as a four-step update:

1. QK^T computes how strongly each token should look at each other token.
2. Division by $\sqrt{d_k}$ keeps the scores from becoming too large.
3. Softmax turns the scores into attention weights a_{ij}, where a_{ij} says how much token i reads from token j.
4. Multiplication by V builds the new token vector as a weighted mixture of value vectors:

$$z_i = \sum_{j=1}^{n} a_{ij} v_j.$$

So the attention result is used directly as an updated representation of each token. In a real Transformer block, this attention output is usually passed through an output projection, added back to the previous token vector through a residual connection, normalized, and then sent through a feed-forward network. A simplified block-level picture is

$$X \to Z_{\text{att}} \to \text{LayerNorm}(X + Z_{\text{att}} W_O) \to \text{feed-forward update}.$$

For this course, the important idea is enough:

$$\text{old token vectors} \to \text{attention} \to \text{context-updated token vectors}.$$

ELI5. For token i, attention creates a customized summary of the other tokens. It does not choose one word only. It assigns weights to words and mixes their value vectors into a new vector z_i.

Common trap. Do not say that attention "just computes attention scores". The scores are only an intermediate step. After softmax, the weights are multiplied by the value vectors, producing new contextual token vectors that continue through the Transformer block.

11.7.6 Positional information

Referenced concepts. Uses: embeddings (Section 9.6); Transformer (Section 11.7); query/key/value attention (Section 11.7.4).

Self-attention alone does not know word order. Therefore positional encodings or positional embeddings are added to token embeddings before attention is computed.

The simplest way to write this is

$$x_i = e_i + p_i,$$

where

- e_i is the token embedding of the token at position i;
- p_i is the positional vector for position i;
- x_i is the input vector that is actually sent into the Transformer block.

For example, suppose the token embedding is

$$e_i = [0.20, -0.10, 0.70]$$

and the positional vector for its position is

$$p_i = [0.01, 0.04, -0.02].$$

Then the Transformer starts from

$$x_i = e_i + p_i = [0.21, -0.06, 0.68].$$

This means that the same word has almost the same lexical meaning vector, but the model can still distinguish whether it appeared early or late in the sequence. After this addition, Q, K, and V are computed from x_i, so position influences attention scores and token updates.

Some Transformers use learned positional embeddings; others use fixed formulas such as sinusoidal encodings or newer relative/rotary variants. For the exam-level idea, it is enough to know that token identity and token position are combined before self-attention.

Common trap. Do not say that a Transformer ignores order. More precise: raw self-attention is permutation-invariant/equivariant, so the model needs explicit positional information.

11.7.7 Transformer versus RNN

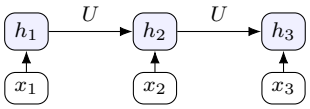
Referenced concepts. Uses: RNN (Section 11.5); Transformer (Section 11.7); attention (Sections 11.7.4, 11.7.5); positional information (Section 11.7.6).

The easiest comparison is graphical. For the same three-token input, an RNN creates a chain of hidden states. Token 3 only receives information about token 1 through the intermediate hidden states. A Transformer updates token vectors with attention: token 3 can directly read information from token 1 inside one self-attention layer.

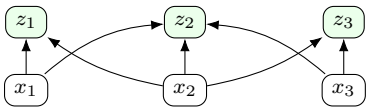
In the diagrams below, x_i is the input token vector. In the RNN, the updated representation is the hidden state h_i. In the Transformer, the updated representation is the attention output z_i. Both are contextual representations, but they are computed in different ways.

The diagrams are simplified, but they show the key difference in information flow.

RNN: hidden-state chain



Transformer: attention links



Feature	RNN	Transformer
Processing	Sequential: token t needs hidden state from token $t-1$.	Parallel over positions during training: token representations in a layer can be computed together.
Context mechanism	Hidden state h_t carries compressed past information.	Self-attention mixes information from other token representations.
Order	Natural from sequence processing.	Added by positional embeddings/encodings.
Long dependencies	Hard in simple RNNs because information must survive many state updates.	Easier because attention can directly connect distant tokens, though cost grows with sequence length.
Learned weights	Reuses U, W, V across time steps.	Learns projection matrices W_Q, W_K, W_V plus feed-forward layers in each block.
Main cost issue	Slow sequential computation.	Attention is quadratic in sequence length.

Exam use. A good Transformer exam answer should mention: token embeddings, positional information, self-attention, Q/K/V, feed-forward layers, contextual representations, and next-token prediction.

Chapter 12

Exam-Oriented Study Guide

12.1 Highest-priority topics

> **Referenced concepts.** Used here: return G_t (Section 1.4); discount factor γ (Section 1.5.1); state-value function V, V^π (Sections 1.5, 1.11); Bellman equation (Sections 1.6, 2.1); MDP (Section 1.7); policy π (Section 1.9); Monte Carlo evaluation (Section 3.2).

Based on the model exam and uploaded materials, the safest high-priority list is:

1. RL basics: MRP (Section 1.3), MDP (Section 1.7), return (Section 1.4), discount factor (Section 1.5), value functions (Sections 1.5, 1.11), Bellman equations (Sections 1.6, 2.1).
2. Model-free evaluation: Monte Carlo (Section 3.2) and TD updates (Section 3.3).
3. Model-free control: SARSA (Section 4.5), Q-learning (Section 4.6), DQN (Section 5.8), on-policy/off-policy (Section 4.3), ε-greedy (Section 4.2), and GLIE (Section 4.8).
4. Bandits: regret and action values (Section 6.2), greedy failure, ε-greedy limitations, UCB, Bayesian updating.
5. COLT: concept learning (Section 7.2), mistake bound (Section 7.4), k-CNF idea, PAC (Section 7.7), VC dimension (Section 7.8).
6. NLP: Naive Bayes (Section 8.7), n-grams (Section 8.5) and perplexity (Section 8.6), TF-IDF (Section 9.3), embeddings (Section 9.6)/skip-gram (Section 9.5), topic models (Sections 10.2 and 10.6), RNNs (Section 11.5), Transformers (Section 11.7).

12.2 Tutorial and model-exam question bank

> **Referenced concepts.** Used here: MDP (Section 1.7); Bellman equations (Sections 1.6, 2.1); TD learning (Section 3.3); learning-rate conditions (Section 3.5); SARSA (Section 4.5); Q-learning (Section 4.6); DQN (Section 5.8); Naive Bayes (Section 8.7); embeddings (Section 9.6); Transformers (Section 11.7); COLT (Chapter 7).

This section collects the important question types from the uploaded tutorial sheets and model-exam-style materials. It is not meant to replace the official sheets; it is a map of what the skripta should prepare you to answer.

Question type	What your answer should contain

Fixed action sequence vs policy	A fixed action sequence is not enough for stochastic MDPs; actions must depend on the observed state. Fixed sequences belong more naturally to classical deterministic planning.
Bellman equation variants	Explain whether reward is written as $r(s)$, $r(s,a)$, or $r(s,a,s')$. The core idea is always: value now equals immediate reward plus discounted value of neighboring states.
Discount factor questions	Know $\gamma \in [0,1]$, what happens when $\gamma = 0$, why high/low γ makes sense in different environments, and why discounting helps infinite-horizon problems.
MC policy evaluation from a sequence	Compute returns-to-go for visits of each state, then average them. Know first-visit vs every-visit.
TD update from a sequence	Apply $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$ step by step. Remember that only the current state's estimate changes at each step.
Learning-rate convergence	State the Robbins-Monro conditions $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$, and give examples such as $1/t$ or $1/N_t(s)$.
Greedy with respect to own value function	Explain that if a policy is unchanged by policy improvement, policy iteration would terminate, so the policy is optimal under the standard assumptions.
GLIE and exploration schedules	Decide whether a schedule has infinite exploration and becomes greedy in the limit. Random forever is not greedy in the limit; stopping exploration too early is not infinite exploration.
SARSA vs Q-learning	Write both updates and explain the target: SARSA uses the action actually sampled next; Q-learning uses the best currently estimated next action.
Adversarially changed action	Q-learning can still learn optimal greedy values because its target uses a max; SARSA learns values for the actually followed behavior. Mention convergence assumptions.
Q-learning table/path update	Given a path and a Q matrix, update only the visited state-action entries using the Q-learning target.
DQN / Deep RL lab questions	Explain that DQN replaces the Q table by a neural network, uses replay buffer D, target network w^-, and target $y = r + \gamma \max_{a'} \widehat{Q}(s', a'; w^-)$.
Bandit regret/UCB	For bandits, $Q(a)$ is expected reward of arm a. Regret is opportunity loss relative to the optimal arm. UCB chooses using estimate plus uncertainty bonus.
Naive Bayes text classification	Write $P(c) \prod_i P(w_i \mid c)$, compute probabilities from counts, classify the review, and mention smoothing/log-space.
TF-IDF	Explain term frequency times inverse document frequency and compute it from counts.
Embeddings/word2vec	Explain dense vectors, one-hot lookup, skip-gram, negative sampling, and why similar words get similar vectors.
Transformers	Explain tokens, embeddings, positional information, self-attention, query/key/value, and the difference from RNNs.

COLT	Explain concept class vs hypothesis class, mistake-bound learning, conjunctions/k-CNF, PAC idea, finite hypothesis class bound, and VC dimension.

Common trap. Bayesian networks are intentionally not covered in this draft. Bayesian ideas are kept only where they are part of bandits or topic models.

12.3 Remaining limitations

Referenced concepts. Used here: features (Section 5.2).

This is still a study draft, not a full official textbook. It does not contain every slide detail, all tutorial solutions, or a complete exercise collection. Some formulas are intentionally presented without proofs. For exam preparation, this is a feature rather than a bug, but for complete formal study you should still consult the original slides.

12.4 Formula and calculation lookup

Referenced concepts. Each formula below includes a clickable pointer to the section where the notation is defined or explained. This is a lookup table, not a proof table: use it to remember what quantity is being computed and where it is used.

Referenced concepts. Used here: transition model P (Sections 1.2, 1.7); reward function R (Sections 1.3, 1.7); return G_t (Section 1.4); discount factor γ (Section 1.5.1); state-value function V, V^π (Sections 1.5, 1.11); policy π (Section 1.9); action-value function Q^π (Section 1.12); DQN (Section 5.8); TD learning (Section 3.3); learning-rate conditions (Section 3.5); PAC (Section 7.7); topic models (Chapter 10); Transformers (Section 11.7).

Takeaway. The reward function R is the expected reward model. A lowercase observed reward such as r_t is one actual reward sample. If the model is unknown, R can be estimated by averaging observed rewards for the same state or state–action pair.

Concept / calculation	Formula / idea
Exact transition model P (Sections 1.2, 1.7)	The exact model gives probabilities such as $P(s' \mid s)$ in an MP/MRP or $P(s' \mid s, a)$ in an MDP. The rows below are only empirical estimates used when P is unknown but transitions have been observed.
Counting notation for empirical estimates	$N(s)$ = number of observed visits to state s as the current state. $N(s, s')$ = number of observed transitions $s \to s'$. $N(s, a)$ = number of times action a was taken in state s. $N(s, a, s')$ = number of observed transitions $s \xrightarrow{a} s'$.
Transition estimate in an MP/MRP (Section 1.8)	Estimate by relative frequency: $\widehat{P}(s' \mid s) = \frac{N(s, s')}{N(s)}$. Meaning: out of all observed visits to s, count what fraction ended in s'.
Transition estimate in an MDP (Section 1.8)	With actions: $\widehat{P}(s' \mid s, a) = \frac{N(s, a, s')}{N(s, a)}$. Meaning: out of all observed times where action a was taken in state s, count what fraction ended in s'.

Exact reward function R (Sections 1.3, 1.7)	The exact reward model is an expectation: $R(s) = \mathbb{E}[R_t \mid X_t = s]$ in an MRP, or $R(s,a) = \mathbb{E}[R_t \mid X_t = s, A_t = a]$ in an MDP. The rows below estimate this expectation by averaging observed reward samples.
Reward estimate in an MRP (Section 1.8)	From samples: $\widehat{R}(s) = \frac{1}{N(s)} \sum_{t:s_t=s} r_t$. Meaning: average all observed rewards received when the current state was s.
Reward estimate in an MDP (Section 1.8)	From samples: $\widehat{R}(s,a) = \frac{1}{N(s,a)} \sum_{t:s_t=s,a_t=a} r_t$. Meaning: average all observed rewards received after taking action a in state s.
Policy-induced MRP (Section 1.10)	For a fixed policy π: $P^\pi(s' \mid s) = \sum_a \pi(a \mid s) P(s' \mid s, a)$ and $R^\pi(s) = \sum_a \pi(a \mid s) R(s,a)$.
Discount factor (Section 1.5.1)	$\gamma \in [0,1]$. If $\gamma = 0$, only immediate reward matters; if γ is close to 1, future rewards matter more.
Return (Section 1.4)	$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k>0} \gamma^k r_{t+k}$.
State value in an MRP (Section 1.5)	$V(s) = \mathbb{E}[G_t \mid X_t = s]$: expected return when starting in state s.
Bellman equation for MRP (Section 1.6)	$V(s) = R(s) + \gamma \sum_{s'} P(s' \mid s) V(s')$.
State value under policy (Section 2.1)	$V^\pi(s) = \sum_a \pi(a \mid s) \left[R(s,a) + \gamma \sum_{s'} P(s' \mid s, a) V^\pi(s') \right]$.
Optimal Bellman backup / value iteration (Section 2.4)	$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s' \mid s, a) V_k(s') \right]$.
Action value (Section 1.12)	$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s' \mid s, a) V^\pi(s')$. It evaluates taking action a first, then following π.
Greedy policy improvement (Section 2.2)	$\pi'(s) \in \operatorname{argmax}_a Q^\pi(s,a)$. Choose the action with highest state–action value.
Monte Carlo value estimate (Section 3.2)	Average observed returns after visits to s: $\widehat{V}(s) = \frac{1}{N(s)} \sum_{i:s_i=s} G_i$.
TD update (Section 3.3)	$V(s) \leftarrow V(s) + \alpha\,(r + \gamma V(s') - V(s))$; $V(s')$ is a current estimate, so TD bootstraps.
Learning-rate convergence (Section 3.5)	A common sufficient condition is $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$. Example: $\alpha_t = 1/t$.
ε-greedy action choice (Section 4.2)	With probability $1 - \varepsilon$, choose a greedy action; with probability ε, choose a random action.
SARSA (Section 4.5)	$Q(s,a) \leftarrow Q(s,a) + \alpha\,(r + \gamma Q(s',a') - Q(s,a))$; uses the action actually chosen next.
Q-learning (Section 4.6)	$Q(s,a) \leftarrow Q(s,a) + \alpha\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$; uses the best currently estimated next action.
DQN target (Section 5.8)	For non-terminal s': $y = r + \gamma \max_{a'} \widehat{Q}(s', a'; w^-)$. Train $\widehat{Q}(s,a;w)$ toward y.
Bandit action value (Section 6.2)	$Q(a) = \mathbb{E}[R_t \mid A_t = a]$. In bandits there is only one state, so no s is needed.
Bandit regret (Section 6.2)	$V^* = \max_a Q(a)$, $L_t = V^* - Q(A_t)$, $L_T^{tot} = \sum_{t=1}^T L_t$.
UCB score (Section 6.5)	One common form: $U_t(a) = \widehat{Q}(a) + \sqrt{\log(t)/(2N(a))}$; choose $a_t \in \operatorname{argmax}_a U_t(a)$.
Beta–Bernoulli update (Section 6.6)	If $\theta \sim \text{Beta}(\alpha, \beta)$ and we observe p successes, n failures, then $\theta \mid \mathit{data} \sim \text{Beta}(\alpha + p, \beta + n)$.
Mistake-bound halving (Section 7.5)	If the version space has size $\lvert H \rvert$, the halving algorithm makes at most $\lceil \log_2 \lvert H \rvert \rceil$ mistakes.
Finite-class PAC bound (Section 7.7.5.1)	For a consistent learner over finite H, a typical bound is $m \geq \frac{1}{\varepsilon} \ln \frac{\lvert H \rvert}{\delta}$.
VC/PAC sample bound idea (Section 7.8)	Replace $\ln \lvert H \rvert$ by a capacity term based on $\text{VC}(H)$ when H is infinite or very large.
Naive Bayes text classification (Section 8.7)	$\widehat{c} = \operatorname{argmax}_c P(c) \prod_i P(w_i \mid c)$; in practice use logs to avoid underflow.
Bigram MLE (Section 8.5)	$P(w_i \mid w_{i-1}) = \text{count}(w_{i-1}, w_i) / \text{count}(w_{i-1})$.
Perplexity (Section 8.6)	For N tokens: $PP = \exp\left(-\frac{1}{N} \sum_{i=1}^N \log P(w_i \mid \mathit{context})\right)$. Lower is better.
Cosine similarity (Section 9.2.3)	$\cos(x,y) = \frac{x \cdot y}{\lVert x \rVert \lVert y \rVert}$; often used for sparse or dense vector comparison.
TF-IDF (Section 9.3)	$\mathit{tfidf}(t,d) = \mathit{tf}(t,d)\,\mathit{idf}(t)$. High if a term is frequent in a document but not common everywhere.
LSA / truncated SVD (Section 10.3)	Apply SVD to a document–term matrix X: $X \approx U_k \Sigma_k V_k^\top$. This gives latent semantic dimensions.
NMF (Section 10.4)	Factorize a non-negative matrix: $X \approx WH$, where $W, H \geq 0$. Often used for interpretable topic-like components.
LDA topic idea (Section 10.5)	Each document has a topic mixture θ_d; each topic is a distribution over words ϕ_k; hidden word-topic assignments are inferred from data.

RNN update (Section 11.5)	$h_t = f(Wx_t + Uh_{t-1} + b)$; h_t is the computed hidden state carrying past information.
Transformer attention (Section 11.7.5)	$\text{Attention}(Q, K, V) = \text{softmax}(QK^\top/\sqrt{d_k})V$. The result is an updated token representation.
Positional embedding (Section 11.7.6)	A token representation can be $x_i = e_i + p_i$, where e_i is token embedding and p_i is position embedding.

12.5 Practice prompts

Referenced concepts. Used here: discount factor γ (Section 1.5.1); TD learning (Section 3.3); learning-rate conditions (Section 3.5); SARSA (Section 4.5); Q-learning (Section 4.6); Naive Bayes (Section 8.7).

Exercise 12.1.
Given $V(s) = 4$, $r = 1$, $V(s') = 10$, $\gamma = 0.5$, and $\alpha = 0.2$, perform one TD update.

Exercise 12.2.
Explain in two sentences the difference between SARSA and Q-learning.

Exercise 12.3.
Explain the role of the replay buffer and target network in DQN.

Exercise 12.4.
Classify a two-class document using Naive Bayes and explain why smoothing is needed.

Exercise 12.5.
Explain why a Transformer needs positional information.

Historical Orientation and Modernity Notes

This table is a study-orientation guide, not a precise history exam. "Central period" means the time when the method was especially important or commonly treated as a standard approach. "Modern successor" means what usually plays the central role today, not that the older method is useless.

The final column is a rough estimate of how much of that topic is mostly historical/foundational rather than the current default method for the same practical task. It is intentionally approximate and is meant to answer the study question: "Am I learning this because it is still the main method, or because it explains the foundations?" It is not a measured percentage of lecture time and not a claim about industrial usage statistics.

Rough course-level estimate	Approximate share
Foundational/classic material that is not usually the current default production method for the same practical task	about 65–75%
Current methods or directly modern bridges, such as DQN, bandits in applications, hybrid retrieval, contextual embeddings, and Transformers	about 25–35%

This does not mean the older material is useless. Some older methods are direct predecessors of modern ones: tabular Q-learning leads naturally to DQN. Other methods are more historically separate: LSA and LDA help explain interpretable corpus analysis, but they are not the core mechanism of modern Transformer LLMs.

Method or topic	Central period	What mostly replaced it or complements it today	Rough modernity note
Dynamic programming for MDPs	classic AI/RL, 1950s–1990s foundations	Still fundamental theory; large unknown environments use model-free RL, approximate RL, and deep RL.	Mostly foundational (about 70–85%), but still needed to understand RL.
Monte Carlo and TD evaluation	classic RL, 1980s–1990s onward	Still core RL ideas; in large problems they are combined with function approximation.	Foundational but not orphaned (about 50–70%).
SARSA and tabular Q-learning	1990s tabular RL	DQN and later deep RL methods when states are large or continuous.	Foundational predecessor (about 65–80%); directly explains DQN.
DQN	around 2013–2015 deep RL breakthrough period	More advanced deep RL variants, actor-critic methods, policy-gradient methods, model-based RL.	Older deep-RL baseline (about 40–60%); still important bridge.
Bandit algorithms such as UCB/Thompson sampling	classic and still active	Still used directly in online decision making, recommendation, ads, and experimentation.	Still practically relevant (about 20–40% historical).
COLT, PAC, VC dimension	1980s–1990s learning theory foundations	Still theoretical foundations; modern deep learning needs additional generalization theory.	Mostly foundational theory (about 70–90%).
N-gram language models	1990s–2000s practical statistical NLP	Neural language models and Transformers.	Mostly historical for LMs (about 85–95%), but useful for probability intuition.

Naive Bayes for text	1990s–2000s simple text classification	Logistic regression/SVMs, then neural encoders and Transformer classifiers; still useful as a baseline.	Mostly baseline/foundation (about 70–85%).
TF-IDF and sparse vector retrieval	1970s onward; strong baseline for decades	Still used in search, often combined with dense neural embeddings in hybrid retrieval.	Not obsolete (about 30–50% historical).
LSA/SVD	1990s latent semantic indexing	Neural embeddings for semantic representation; LSA still useful for explaining matrix methods.	Mostly historical for semantic NLP (about 80–90%).
NMF topic extraction	late 1990s–2000s interpretable matrix factorization	Still useful for interpretable topics; often complemented by embedding-based clustering/topic discovery.	Classic but usable (about 50–70% historical).
LDA topic modeling	2000s–early 2010s probabilistic topic modeling	Embedding-based topic discovery and Transformer representations for many applications; LDA remains useful for interpretable corpus exploration.	Classic but still interpretable (about 60–75% historical).
word2vec/static embeddings	2013–2017 neural NLP standard representation	Contextual embeddings such as BERT-style models and modern LLM token representations.	Important predecessor (about 60–75% historical).
RNN/LSTM sequence models	1990s–2017, especially before Transformers	Transformers for most large-scale NLP; RNNs still appear in smaller/sequential systems.	Mostly replaced in large-scale NLP (about 75–90%).
Transformers	2017 onward	Still central for modern LLMs; many current improvements modify attention, scaling, retrieval, or training recipes.	Current core method (about 0–20% historical).

Takeaway. A rough reading of the course is: much of RL and NLP is taught through older or foundational methods because they explain the logic of the modern methods. Q-learning is not orphaned because DQN is basically Q-learning with a neural approximator and stabilization tricks. In contrast, LSA/LDA are more historically separate from modern LLMs: they are useful for interpretable corpus analysis, but they are not the mechanism that powers Transformer LLMs.